Titre: Title:	On Run-Time Configuration Engineering
Auteur: Author:	Mohammed Sayagh
Date:	2018
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Sayagh, M. (2018). On Run-Time Configuration Engineering [Thèse de doctorat, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/3182/

UNIVERSITÉ DE MONTRÉAL

ON RUN-TIME CONFIGURATION ENGINEERING

MOHAMMED SAYAGH
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
JUILLET 2018

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

ON RUN-TIME CONFIGURATION ENGINEERING

présentée par : SAYAGH Mohammed
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

M. MULLINS John, Ph. D., président
M. ADAMS Bram, Ph. D., membre et directeur de recherche
M. KHOMH Foutse, Ph. D., membre
M. KÄSTNER Christian, Doktor-Ingenieur, membre externe

DEDICATION

To my mother, father, wife, and whole family

ACKNOWLEDGEMENTS

I would like to first express my sincere thanks to my supervisor Dr. Bram Adams for his exceptional support of my Ph.D and for his motivation and valuable feedbacks.

Special thanks to all my co-authors for their help and contributions: Dr. Noureddine Kerzazi, Dr. Fabio Petrillo, Dr. Artur Andrzejak, Dr. Zhen Dong, and Khalil Bennani.

I would like to thank my thesis committee Dr. John Mullins, Dr. Christian Kästner, and Dr. Foutse Khomh for accepting my invitation to be jury members, Dr. Jason Robert Tavares to be representative of my defense.

Last but not least, I also would like to thank all my labmates who created such a positive working atmosphere: Ruben, Fabio, Mohab, Rodrigo, Amine, Armstrong, Azadeh, Mbarka, Aminata, and Le. Thanks also for your constructive and inspiring talks in the lab.

RÉSUMÉ

De nos jours, les utilisateurs changent le comportement de leur logiciel et l'adaptent à différentes situations et contexts, sans avoir besoin d'aucune modifications du code source ou recompilation du logiciel. En effet, les utilisateurs utilisent le mécanisme de configuration qui offre un ensemble d'options modifiables par les utilisateurs.

D'après plusieurs études, des mauvaises valeurs des options de configuration causent des erreurs difficiles à déboguer. Plusieurs compagnies importantes, comme Facebook, Google et Amazon ont rencontré des pannes et erreurs sérieuses à cause de la configuration et qui sont considérées parmi les plus pires pannes dans ces compagnies. En plus, plusieurs études ont trouvé que le mécanisme de configuration augmente la complexité des logiciels et les rend plus difficile à utiliser. Ces problèmes ont un sérieux impact sur plusieurs facteurs de qualité, comme la sécurité, l'exactitude, la disponibilité, la compréhensibilité, la maintenabilité, et la performance des logiciels.

Plusieurs études ont été élaborées dans des aspects spécifiques dans l'ingénierie des configurations, dont la majorité se concentrent sur le débogage des défaillances de configuration et les tests de la configuration des logiciels, tandis que peu de recherches traitent les autres aspects de l'ingénierie des configurations de logiciel, comme la création et la maintenance des options de configuration. Par contre, nous pensons que la configuration des logiciels n'a pas seulement un impact sur l'exactitude d'un logiciel, mais peut avoir un impact sur d'autres métriques de qualité comme la compréhensibilité et la maintenabilité.

Dans cette thèse, nous faisons d'abord un pas en arrière pour mieux comprendre les activités principales liées du processus de l'ingénierie des configurations, avant d'évaluer l'impact d'un catalogue de bonnes pratiques sur l'exactitude et la performance du processus de la configuration des logiciels. Pour ces raisons, nous avons conduit un ensemble d'études empiriques qualitatives et quantitatives sur des grands projets libres. On a conduit une étude qualitative en premier lieu, dans laquelle nous avons essayé de comprendre le processus de l'ingénierie de configuration, les enjeux et problèmes que les développeurs rencontrent durant ce processus, et qu'est ce que les développeurs et chercheurs proposent pour aider les développeurs à améliorer la qualité de l'ingénierie de la configuration logiciel. En réalisant 14 entrevues semi structurées, un sondage et une revue systématique de littérature, nous avons défini un processus de l'ingénierie de configuration invoquant 9 activités, un ensemble de 22 challenges rencontrés en pratique et 24 recommandations des experts.

Pour mieux comprendre comment les développeurs gèrent leurs configurations dans le code source, on a élaboré une étude quantitative en deuxième étape pour mieux comprendre l'utilisation des cadriciels dédiés à la configuration. Étonnamment, on a trouvé que malgré l'existence de cadriciels de configuration très sophistiqués, 47% des projets que nous avons analysés ne les utilisent pas, et suivent simplement une approache ad-hoc, tandis que les cadriciels les plus basiques sont les plus populaires. Pour aider les développeurs à choisir un bon cadriciel, on a élaboré une taxonomie de fonctionnalités à considérer, et on a construit un modèle explicatif pour prioriser les fonctionnalités les plus importantes qui peuvent avoir un impact sur les efforts nécessaires pour maintenir la configuration d'un logiciel.

Pour mieux comprendre quelles fonctionnalités un cadriciel devrait avoir pour bien gérer les enjeux importants identifiés dans le processus d'ingénierie de configuration, et ainsi augmenter la probabilité d'adoption par les développeurs, nous avons évalué 4 principes de bonnes pratiques d'ingénierie de configuration que nous avons identifiées dans nos entrevues dans une étude d'utilisateur. Ces principes tournent autour de l'idée de traiter les options de configuration comme du code, en tant qu'utiliser les principes de l'ingénierie logicielle tel que l'encapsulation. Nous avons prototypé ces 4 principes dans un cadriciel appelé Config2Code, que nous avons ainsi comparé avec le cadriciel populaire Preferences à travers une étude utilisateur qui couvre 11 tâches de configuration typiques, avec 55 participants incluant 17 experts en développement de logiciels. Ces principes ont été capables d'améliorer 72% des tâches de configurations.

Malgré que plusieurs efforts de recherche ont été conduite pour l'activité de débogage des défaillances de configuration et trouver quelles options il faut changer pour résoudre une défaillance, ces recherches se concentrent juste sur les erreurs dans des applications uniques. En effet, ces approches ignorent les défaillances de configuration dont la cause vient d'une couche plus profonde de la pile de l'application, par exemple, une défaillance dans une extension de WordPress causée par une valeur incorrecte de la configuration du serveur web. Déboguer de telles erreurs de configuration multi-couches est difficile, parce que chacune de ces couches est un composant séparé qui est développé avec un langage de programmation différent, et qui consomme des services des autres couches. Un exemple connu de cette architecture est la pile LAMP, qui est une pile populaire pour les applications web utilisée pour implémenter plus que 80% de tous les sites web qui existent.

Cependant, dans notre analyse du potentiel d'avoir des erreurs de configuration dans la pile LAMP, nous avons trouvé que jusqu'à 85.16% des options de Wordpress sont utilisées par une couche supérieure, ce qui indique que la modification d'une seule option de WordPress peut avoir un impact sur les autres couches. Une analyse de 1,042 erreurs de configuration

d'une seule couche et de multiples couches démontre que les erreurs de configuration à travers plusieurs couches sont sévères, vu qu'elles arrivent dans des environnements de production et demandent beaucoup d'efforts pour les résoudre.

Finalement, nous avons proposé une approche modulaire pour aider les utilisateurs à déboguer les erreurs de configuration qui arrivent dans les architectures multi-couches. Notre approche combine des techniques d'analyse du code existantes pour trouver les options qui sont mal-configurées. Nous avons évalué notre approche sur 36 erreurs de configuration réelles, et trouvé que notre approche est capable de trouver les options mal-configurées dans quelques minutes, ce qui rend notre approche pratiquement utilisable.

Notre thèse confirme que les développeurs rencontrent un nombre considérable d'enjeux dans plusieurs activités d'ingénierie, et l'application des bonnes pratiques du code à l'ingénierie de configuration aussi améliore la qualité de l'ingénierie des configuration, comme la maintenabilité, compréhensibilité, utilisabilité, disponibilité et l'exactitude. Nous avons aussi trouvé dans notre thèse que les erreures de configuration à travers multicouches sont sévères et difficiles à déboguer, et une approche modulaire qui combine des techniques de l'analyse du code source aide pour débogger de telles défaillances.

ABSTRACT

Modern software applications allow users to change the behavior of a software application and adapt it to different situations and contexts, without requiring any source code modifications or recompilations. To this end, applications leverage a wide range of mechanisms of software configuration that provide a set of options that can be changed by users.

According to several studies, incorrect values of software configuration options cause severe errors that are hard-to-debug. Major companies such as Facebook, Google, and Amazon faced serious outages and failures due to configuration, which are considered as some of the worst outages in these companies. In addition, several studies found that the mechanism of software configuration increases the complexity of a software system and makes it hard to use. Such problems have a serious impact on different quality factors, such as security, correctness, availability, comprehensibility, maintainability, and performance of software systems.

Several studies have been conducted on specific aspects of configuration engineering, with most of them focusing on debugging configuration failures and testing software configurations, while only few research efforts focused on other aspects of configuration engineering, such as the creation and maintenance of configuration options. However, we think that software configuration can not only have a negative impact on the correctness of a software system, but also on other quality metrics, such as its comprehensibility and maintainability.

In this thesis, we first take a step back to better understand the main activities involved in the process of run-time configuration engineering, before evaluating the impact of a catalog of best practices on the correctness and performance of the configuration engineering process. For these purposes, we conducted several qualitative and quantitative empirical studies on large repositories and open source projects. We first conducted a qualitative study, in which we tried to understand the configuration engineering process, the challenges and problems developers face during this process, and what practitioners and researchers recommend to help developers to improve their software configuration engineering quality. By conducting 14 semi-structured interviews, a large survey, and a systematic literature review, we identified a process of configuration engineering involving 9 activities, a set of 22 challenges faced in practice, and a set of 24 recommendations by experts.

To better understand how developers manage their software configuration in the source code, in a second study, we conducted a quantitative analysis to understand the usage of existing frameworks that are dedicated to software configuration options. Surprisingly, we found that despite the existence of very sophisticated configuration frameworks, 47% of our studied

projects do not use them, and follow just an ad-hoc approach, while we found that the most popular frameworks that are used in practice are only basic frameworks. To help practitioners to select a suitable configuration framework, we derived a taxonomy of features to consider, and build an explanatory model to prioritize the most important features that could have an impact on the effort required to maintain software configuration options.

To better understand why existing frameworks might not be adopted, we performed a large user study to evaluate 4 principles of good configuration engineering that we identified from our interviews. The principles center around the idea of treating run-time configuration as code, which allows to benefit from software engineering staples like encapsulation. We implemented these four principles in a framework called Config2Code, which we then compared to the popular Preferences framework via a user study that covers 11 typical configuration tasks, and has 55 participants including 17 software development experts, suggesting that configuration frameworks have the potential to help practitioners, once adopted. The principles were able to improve 72% of configuration engineering tasks.

We then focused on one of the major configuration activities, i.e., debugging software configuration failures, since we observed how despite a large volume of existing work most of the existing approaches focus only on errors in single applications. As such, these approaches ignore configuration failures whose cause is buried in a deeper layer of the application's runtime stack, for example, a failure of a WordPress plugin due to an incorrect configuration value within the web server's configuration. Debugging such cross-stack configuration failures is challenging because each of these layers is a separate component that is developed in a different programming language, and consumes services of other layers. A popular example of such an architecture is the LAMP stack, which is a popular stack of web applications used to implement more than 80% of all existing websites.

Therefore, we empirically analyzed the potential to have configuration errors in the LAMP stack. We found that up to 85.16% of WordPress configuration options are used by another higher layer, which indicates that the modification of a given option's value in WordPress potentially can have a substantial impact on other layers. Furthermore, analysis of 1,042 real single layer and cross-stack configuration errors learns that cross-stack configuration errors are severe, as they occur in production environments and need a lot of effort to be fixed.

Finally, we proposed a modular approach to help users debug cross-stack configuration errors. Our approach composes existing source code analysis techniques to find culprit options. We evaluated our approach on 36 real cross-stack configuration errors, and found that our approach is able to find the misconfigured options within only few minutes, which makes our approach practically useful.

Our thesis confirmed practitioners face a considerable amount of challenges on different engineering activities, and applying source code best practices to configuration engineering can improve configuration engineering quality, such as software maintainability, comprehensibility, usability, availability, and correctness. We also found in our thesis that cross-stack configuration failures are severe and hard-to-debug, and we propose a modular approach that combines existing source code techniques to debug such failures.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ANNEXES

CHAPTER 1 INTRODUCTION

Users often need to change the behaviour of their software systems or adapt them to different situations and contexts. As a basic example, a user might need to enable or disable a software feature, like enabling the feature that saves the navigation history of Firefox. Users might also need to customize more specific behaviours of their software system like changing the expiry date of browser cookies. Software systems are not only adapted by their final customers, they can be adapted by administrators and sysadmins as well. For example, sysadmins can allow only a specific category of users to add new pages to a WordPress based website, or configure the file size limit of attachments to WordPress posts. In addition to these security modifications, administrators can tweak the performance of a software system. For example, they can allow only a certain number of simultaneous connections to a MySQL database.

To achieve such objectives, end users and sysadmins can rely on the mechanism of software configuration, which allows users or operators to customize their software system by just changing the value of configuration options. For example, one only needs to change the value of option *max_execution_time* in the configuration file *php.ini* to limit the maximum time allowed to run a PHP script, or change the option *max_connections* in MySQL configuration file *my.cnf* to limit the number of simultaneous connections to a database, and hence reduce its data load and improve its performance. While these configuration options are run-time options, which do not require users to re-compile the software system, other types of configuration options exist, for example, options whose value is determined during compilation time. In contrast to run-time configuration options, compile-time configuration options are used by developers to make fundamental feature-related decisions within the shipped product, and can also be used to adapt a project to a certain platform. Note that this thesis focuses on run-time configuration options, i.e., options whose value can still be changed after deployment.

While run-time options add significant flexibility for users, they also increase a software system's complexity, which makes it more complicated to configure, understand, maintain, and develop. For example, Firefox has over 2,000 configuration options that allow customizing a large variety of features, while the MySQL database and Apache web server in turn have hundreds of options that sysadmins need to configure and check. For a web app, a sysadmin has to configure not only the MySQL and Apache layers, but also a scripting language, such as the PHP interpreter, the operating system on top of which all of these layers are installed, and the web application itself. Furthermore, this web application can be extended

by installing and configuring plugins. In the end, a sysadmin needs to configure a huge amount of options, each of which belonging to a separate layer of the web application's stack, such as the LAMP (Linux, Apache, MySQL, and PHP) stack. One misconfigured option in one layer can cause serious damages, whose symptoms are only visible in another layer, which we refer to as cross-stack configuration errors.

Having a large number of configuration options might negatively impact a variety of other quality factors, such as software correctness, comprehensibility, maintainability, testability, and performance. Yin et al. [215] found that the largest category of severe problems that are reported by customers are caused by configuration errors. Hence, it comes as no surprise that such configuration errors are also reported by major companies, where they have a serious impact. Recently, in 2017, a simple misconfigured option in an Amazon storage service exposed confidential information of millions of customers to unauthorized people. This exposed information did not only include these customers' personal information such as their names or email addresses, but also their financial information like the last four digits of their credit card numbers [1]. Within the same year, personal data of millions of American voters were exposed due to a similar misconfiguration in an Amazon service [2, 3]. Similarly, due to a misconfiguration error, millions of Facebook users were not able to access the website [96] for around 2.5 hours, which is considered in Facebook as one of the worst outages. A well known Google engineer cited that most production errors are caused by configuration errors rather than source code bugs [14], and described software configuration as "*configuration hell*".

While configuration errors could cause serious damage, testing all possible configurations for a software program is not a feasible solution. Indeed, a typical configurable software system with only one hundred boolean options needs to generate and execute 2^{100} tests, which require practically enormous execution resources. More options and complicated option types, such as hierarchical nested options, require for sure much more tests to write and execute. This is why a large body of research literature proposed approaches to help debug and fix configuration errors [44, 46, 47, 68, 69, 147, 217–219], while a second large category of research efforts have been conducted on software configuration testability [38, 60, 89, 105, 115, 131, 142, 143, 143, 144, 173, 175, 176, 209].

Unfortunately, software configuration options do not only have a negative impact on the correctness and testability of a software system, they can also impact other quality factors, such as software understandability. A highly configurable software system ideally requires users to understand each single option, and which correct and optimal values that option requires. Hence, understanding hundreds of options is not straightforward. While users can

rely on a software system's documentation, such documentation might not always be up-to-date or contain sufficiently clear explanations. Therefore, software configuration might have a negative impact on software usability and comprehensibility as well.

Since adding more configuration options increases a software system's complexity, we need to better understand how developers create new configuration options and why developers keep adding new options although that might have a negative impact. It is also required to understand why developers do not remove software configuration options although users do not consume all existing options [212], and how developers maintain their software configuration options. As such, debugging and testing software configuration errors are just two activities out of the, as yet poorly studied, overall process of configuration engineering that can have a negative impact on the quality of a software system, such as an application's software usability, understandability, and maintainability. It is also necessary to better understand what practices and techniques can improve the process of creating and maintaining a software system's configuration options.

1.1 Thesis Hypothesis

While configuration options offer flexibility to users, they potentially make a software system complex and can cause errors that are common, severe, and hard to debug. Therefore, it is essential to obtain a better understanding of the engineering process of creating and maintaining configuration options, as well as the challenges that are faced at each activity of that process. Hence, we formulate our research hypothesis as follow:

> We hypothesize that (1) configuration engineering is constituted by a wide range of activities for which developers today face a variety of challenges. However, several of the technical challenges can be addressed by a careful selection of (2) source code best practices. For the specific activity of configuration debugging, (3) cross-stack configuration errors are more difficult to debug than single-layer configuration errors, however, they can be debugged effectively by a composition of source code analysis techniques.

To validate this hypothesis, (1) we first took a step backward to understand the process of configuration engineering, the challenges that are faced by practitioners during that process,

and recommendations from practitioners and research literature to improve software configuration engineering quality. We then quantitatively studied the usage of frameworks dedicated to software configuration in open source projects. After this first main study, we took two additional main research directions, in which (2) we evaluated existing source code best practices to address a large set of the technical challenges of software configuration engineering. These practices are based on four principles that were derived iteratively during interviews with practitioners. (3) For the specific configuration activity of configuration debugging, we then evaluated how a modular composition of existing source code analysis techniques is able to debug software configuration errors that span several layers of a stack. The following section details our contributions.

1.2 Thesis Contributions

1.2.1 Qualitative Study to Understand the Configuration Engineering Process, Challenges, and Recommendations

Configuration errors are one of the most common problems in software engineering. To better understand what leads to such problems, and why software configuration can have such a negative impact on software quality, we conducted a qualitative study in which we addressed our first thesis hypothesis. In this qualitative study, we identified engineering practices and challenges related to software configuration.

To achieve this goal, we conducted 14 semi-structured interviews with software engineering experts, having different roles in their context, followed by a large survey with 229 responses, and a systematic literature review.

Following a card sort technique at the end of each of these three steps, we enumerated 9 activities that form the process of software configuration engineering. That process starts by the creation of a new configuration option, deciding which data format new options should have, where they should be defined (in a file, a database, or in any other existing storage mechanism) and how new options should be documented, up to the maintenance of configuration options.

For each of these 9 activities, we enumerated a set of challenges that we identified during our qualitative analysis. Most surprisingly, we found that developers do not plan configuration options, configuration options are added ad hoc. We also found that developers ignore to review newly created options, which lets low quality configuration options slip through to production, in the form of options, such as options with unclear names, wrong default values, or completely undocumented options. Furthermore, we found that developers are not able

to refactor configuration options, even for options that are not used anymore by customers or within the software source code.

For each of these 9 activities, we also enumerated a set of recommendations to improve software configuration quality. For example, one of these recommendations is to allow only few experts to manage software configurations, by allowing which option can be added, and by also managing the refactoring of options.

Our systematic literature review revealed that most research conducted on software configuration focuses on debugging configuration errors and testing software configuration, leaving large opportunities for future work to cover other configuration engineering activities. Our research literature survey guides practitioners on existing techniques and researchers on less covered research directions.

1.2.2 Using Mining Software Repositories to Understand the Usage of Software Configuration Frameworks

To understand challenges faced during software configuration engineering (our first thesis hypothesis), we also quantitatively analyzed the usage of frameworks dedicated to software configuration in the source code. The goal of this contribution is to better understand the way in which configuration-related code is handled in the source code of open source projects, and guide developers to choose good configuration frameworks.

Therefore, we quantitatively studied the popularity and usage of 11 configuration frameworks in around 2000 open source projects. We found that basic configuration frameworks (i.e., provided by the JDK) are the most popular ones, and often complemented by other configuration frameworks that propose more sophisticated features.

We proposed a taxonomy of features to help developers choose a suitable configuration framework. Examples of these features are the quality of documentation, how actively these frameworks are maintained by their developers, and the maturity of these frameworks.

With the same goal of helping developers choose a suitable configuration framework and by quantitatively analyzing the history of open source projects, we studied the impact of each taxonomy feature on the effort required to maintain software configuration options. We found that young and very active configuration frameworks, which have detailed documentation and support hierarchical configuration formats, are the frameworks requiring more maintenance effort.

1.2.3 Four Principles to Improve Software Configuration Engineering Quality

During our qualitative study, we identified that developers typically do not apply good source code practices such as code review to configuration. This is because practitioners consider configuration as an external artifact.

Inspired by our first qualitative study and to validate our second thesis hypothesis, we iteratively developed and evaluated 4 main principles that help developers to consider run-time configuration as code and hence improve the quality of software configuration engineering, in terms of maintainability, understandability, and correctness. These 4 principles are:

- Run-time configuration-as-Code: This principle consists of bringing the declaration of configuration options inside the source code, which allows to explicitly define configuration option access within source code.
- Encapsulation of configuration access: Instead of reading configuration options from different source code locations, this principle consists of focusing the reading points of configuration options within a single reader class, and allowing other client classes to use options only via that reader class.
- Generation of configuration media: Keeping source code and configuration files synchronized is not trivial, and this leads developers to leave dead options in their configuration files. This principle consists of automatically generating the configuration file from code of a software system, and hence automatically removing dead options.
- Automatic validation: The goal of this principle is to automatically validate users' configuration choices, and automatically validate that configuration options respect a set of rules and naming conventions. This guarantees the correctness of a software system and comprehension of configuration options.

We prototyped these principles in a Java framework called Config2Code. Then, we evaluated the impact of these principles via a user study, in which we compared the usage of Config2Code against Preferences, i.e., a popular Java configuration framework that does not implement these principles.

55 subjects participated in our user study, in which we evaluated the correctness and time required to achieve different configuration engineering tasks. Our participants performed 11 tasks, from the creation, maintenance, comprehension, and code review of options, to the debugging of configuration failures.

We found that our approach is able to significantly improve the correctness and time required to achieve 8 out of 11 configuration tasks, while no statistically significant difference is observed in the 3 remaining tasks.

While these principles can help prevent configuration errors to some extent, the next section considers an approach to debug more complex configuration errors where prevention has not worked.

1.2.4 Cross-stack Configuration Errors

While the previous contributions aim at helping developers improve software configuration usability, correctness, comprehensibility, and maintainability, this contribution aims at helping users to debug cross-stack configuration errors. This is an advanced form of configuration error whose symptoms occur in one layer of the architecture, while the root cause of the error occurred in another layer. For example, a WordPress user is not able to upload a file because the PHP configuration option memory_limit is mis-configured.

To address our third hypothesis, we first need to understand the extent to which such cross-stack configuration errors exist, then propose an approach to debug them. We addressed these problems and this research hypothesis via the following two contributions:

Analyzing the Potential of Cross-stack Configuration Errors

To better understand the impact of a software system's configuration options on other layers, we empirically studied the configuration of 15 WordPress plugins, WordPress, and PHP-interpreter configuration options, which together represent the top 3 layers of the LAMP stack.

In this study, we investigated the occurrence of the configuration options in the top two layers of the LAMP stack, then analyzed configuration options evolution across history. We found that WordPress and its 15 studied plugins have an average of 76 configuration options, which are stored in two different storage mediums (database and configuration files). This number is increasing across the history of WordPress and its plugins.

We also studied the impact of WordPress configuration options on its plugins, and the impact of PHP interpreter options on WordPress and its plugins. While WordPress plugins use 1.49% to 9.49% and 1.38% to 15.18% of all options stored respectively in the database and files, 78.88% and 85.16% of database and configuration file options are shared between at least two WordPress plugins. Hence, assigning a wrong value to one WordPress configuration

option could not only impact WordPress, but also its installed plugins. These findings warn practitioners and researchers about the potential impact of cross-stack configuration errors.

We also analyzed the prevalence of cross-stack configuration errors in real cases, by qualitatively and quantitatively analyzing 1,048 StackExchange configuration problems. Via this analysis, we found that cross-stack configuration errors are severe problems, as they often occur in production. We complement the research literature, which found that single layer configuration errors require substantial effort to be fixed, by finding that cross-stack configuration errors require similar amounts of effort. While we found that cross-stack configuration errors require fixing less options than single layer configuration errors, those options are spread across several layers, going as deep as the operating system! Therefore, cross-stack configuration errors require researchers to propose and evaluate approaches to debug such errors.

Extending Source Code Analysis Techniques to Debug Cross-stack Configuration Errors

We then proposed an approach based on source code analysis that takes as input a configuration error symptom, and recommends as output a set of options that are most likely to be misconfigured.

Our approach is modular as it composes existing source code analysis techniques, such as static and dynamic slicing. Our approach separately executes a source code analysis technique on each layer of the LAMP stack, then connect the results via a set of physical links. These physical links represent how each layer is connected to its lower layer. For example, to upload a file, WordPress uses the PHP function "move_uploaded_file", whose implementation is within the PHP-interpreter source code (file "ext/standard/basic_function.h"). In this case, our physical links connect the PHP call to "move_uploaded_file" to its implementation in the PHP interpreter source code.

We evaluated this approach on a set of 36 real cross-stack misconfigurations, and found that our approach is able to accurately find misconfigured options, within an acceptable execution time of few minutes.

CHAPTER 2 LITERATURE REVIEW

A large body of research has focused on software configuration. However, most of these research efforts focused on debugging configuration errors or testing software configurations, while none of them focused on the full range of software configuration engineering activities. Since challenges faced during these activities can have a negative impact on the quality of software configuration, we studied configuration challenges that are faced by practitioners, with the aim of deeply understanding the process of creating and maintaining software configuration options.

Software configuration is the mechanism that allows users to adapt a software system to different situations and contexts, by changing the values of a set of available configuration options. A configuration option consists of a key and a value in which the key represents a configuration name, and the value represents the choice of the user for that option. While the representation of an option as a key-value pair is used within all existing storage mechanisms (files, databases, etc.), more complex representations are used as well. For example, often related options are grouped within a category, yielding a hierarchical structure between options and categories. The Apache web server configuration is a well known example of such types of options, in which sysadmins can define a set of rules and permissions (via the options Allow and Deny) for each user or resource.

Developers can decide to store their configuration options in any storage medium. They can use text, json, xml, or any other file formats, while they can even store their options in a dedicated database for software configuration or a specific table within a database. Developers can also just pass the value of configuration options as command line arguments.

In this Chapter, we will introduce the most popular research directions on software configuration and how the existing work is different from our study. An in-depth systematic literature review on software configuration is provided in Chapter 4.

2.1 Empirical Studies on Software Configuration

Many research efforts have been conducted to better understand software configuration, its complexity and errors. Yin et al. [215] analyzed 546 configuration errors in four open source and one commercial project. They found that 27% of customer errors are caused by software configuration and 31% of highly severe errors are caused by software configuration, which represents the largest category of errors marked with a high severity.

Yin et al. [215] also describe different characteristics related to configuration errors, which can help practitioners and researchers better understand configuration errors and their impact. For example, a large percentage of configuration errors cause hard-to-debug problems, such as crashes, hangs, or performance degradations, and between 12.2% and 29.7% of configuration errors are due to inconsistencies between the values of different configuration options.

Arshad et al. [43] empirically studied 281 configuration errors in Java EE servers, and found that more than one third of the reported problems are caused by configuration errors. They also studied these errors and classified them in three categories: parameter errors, compatibility issues, and missing components.

Jin et al. [95] studied three large software systems, and found that there is a need for debugging approaches that consider software developed with multiple programming languages, and approaches to track when a configuration option's value is modified or used in the source code. They found that an option could be modified by a component, and used by a completely different component, which recommends for tools that debug configuration errors in software systems developed with multiple programming languages.

Xu et al. [185] reported on the state-of-the-art on existing debugging configuration errors techniques. They report the lack of approaches that debug errors on multi-component and multi-layer architectures.

While the previous studies focus on analyzing characteristics of configuration errors, Xu et al. [212] empirically studied the usage of configuration options. They found that many configuration options are not used by end users and hence can be removed by developers to reduce software complexity, where many other options can be simplified by changing their types from string to boolean options as an example.

None of these studies focused on the process of configuration engineering as a whole. They instead focus mostly on studying characteristics of configuration errors, and configuration complexity from the users' side. In addition to the process of creating and maintaining options (which includes debugging configuration errors), we qualitatively studied challenges that are faced by practitioners in each activity of the software configuration engineering process.

2.2 Debugging Configuration Errors

Starting from a bug report, different models have been built to predict if a bug report is related to a configuration error [53, 200, 208] and also to predict which option can be misconfigured [200].

Once developers identify that a bug is caused by a configuration error, several approaches have been proposed to help find which option is misconfigured. A first category of approaches [136, 204] searches when a software went from a working to non-working state, and looks at which options were changed during that transition and are likely to be misconfigured.

Other approaches rely on the users' feedback. Wang et al. [192] proposed to find a fix for configuration failure based on previous user experiences for the same failure. Similarly, AutoBash [178] records how users fix a configuration failure, and uses it to propose fixes for configuration failures.

Another category of work on debugging configuration failures relies on source code analysis [44, 46, 47, 68, 69, 147, 217–219]. These approaches consist of automatically analyzing the source code to recommend which option should be changed. For example, Dong et al. [69] used backward slicing starting from the error line, and forward slicing starting from the configuration reading points, then report options that are in the intersection between these two slices as options that are likely to be misconfigured.

While these approaches help users find which configuration option is misconfigured, other approaches recommend correct values for these misconfigured options. Swanson et al. [181] used Firefox configuration constraints and a sampling algorithm to propose fixes, where Xiong et al. [210, 211] proposed an approach that generates a range of correct values from software configuration constraints, defined as a set of rules that each option should respect.

While the previous approaches help debuggers find which options are misconfigured and what are their correct values, other existing work helps developers to prevent configuration errors. Zhang et al. [221] proposed an approach that injects configuration faults to check how the software system reacts to these faults.

The research literature provided on debugging software configuration focuses only on single-layer applications, while there is a need for approaches that consider multi-components and layers [185], such as the LAMP stack. In our research, we studied the prevalence of cross-stack configuration errors, and evaluated an approach to help developers to debug such failures.

This subsection introduced related work on debugging configuration errors, but we refer to Chapter 4 for more detailed discussion of this related work.

2.3 Testing Software Configuration

Because testing all possible configurations of a highly configurable software system require a lot of execution resources, many research efforts have been conducted to help developers optimize their tests.

A first category of research efforts focused on sampling algorithms [38, 60, 89, 115, 142–144, 173, 209] that optimize the number of tests to run. For example, Cohen et al. [60] used Combinatorial Interaction Testing algorithm. Other approaches [105, 131, 143, 175, 176] introduced source code analysis techniques to reduce the number of options to test. For example, Kim et al. [105] identify which options were used by a test and hence which a developer should focus on.

Apart from approaches that minimize the number of tests, Meinicke et al. [118] and Reisner et al. [153] found that the interaction between configuration options is low, which indicates that the number of options sharing the same source code area is low, and hence there is no need to test all possible combinations of configuration options.

To better understand the impact of a configuration option in order to better maintain options in the source code, easily debug configuration errors, test configuration options, and easily document software configuration, existing approaches [67, 148, 224, 224] have been proposed to map each configuration option to its usage in the source code. Lillack et al. [113, 114] proposed an approach that finds under which configuration option a code fragement can be executed.

Apart from testing software configurations, few strategies were evaluated to prevent configuration problems and improve a software configuration engineering quality. In our work, we evaluated the impact of different best practices on the quality of software configuration engineering, including its usability, correctness, comprehensibility, and maintainability.

2.4 Finding Optimal Configuration Values

A fourth research direction covered by the literature is finding an optimal configuration that guarantees a certain level of software performance. Different strategies were proposed to sample the configuration space for optimal software configurations [58, 66, 70, 81, 109, 134, 135, 156, 170, 171, 183, 207, 223], which are discussed in details in Chapter 4.

While it is important for end users to find an optimal configuration value that respects their performances requirements, this category of research focuses on helping users to correctly configure their software system. This direction can also be used by developers to decide good default values for their software configuration options at creation time. However, we found that deciding about default or optimal configuration values is only one configuration engineering activity among other ones that can have an impact on software configuration engineering quality.

2.5 Software Product Lines and Non Run-time Configuration

Closely related to the field of software configuration, a large volume of work exists on product lines. A product line, also called product family, is a set of tools, practices, and techniques used to create a set of different products from a single software system [42,141,190]. Typically, the architecture of a product line is based on a central platform modeling the common functionalities across products, together with a configurable extension facility able to model variability across products. The engineering process of a product line comprises two steps, i.e., domain engineering and application engineering. Domain engineering comprises the analysis and modeling of common and variable functionality within the domain of the resulting system, for example the domain of operating systems (Linux kernel) or the domain of mobile phones (Android vendor). Application engineering then comprises all activities needed to configure, build and manage concrete products on top of the common platform. For example, the Linux operating system product line allows concrete kernels to be built for desktop systems, gaming consoles, smart devices or even cars!

To model and configure variability in product lines, a wide variety of implementation technologies can be used, from condition compilation to aspect or feature oriented programming, to even elaborate domain-specific languages. The most studied product line technology in academic literature is the C preprocessor. Developers can use the C directive #define to declare a configuration option and check if an option is defined or its value respects a constraint via the directives #ifdef and #if.

Liebig et al. [111] analyzed 40 software product lines to understand the usage of the C preprocessor to implement software product line variability, and how it can impact software comprehension and refactoring. They found that on average 23% of the source code of the projects they analyzed is variable.

Hubaux et al. [87] conducted a survey among Linux and eCos users to understand challenges faced by these users during the configuration of Linux and eCos. They found that there is a lack of documentation and guidance provided by these software systems' configurators.

To help users correctly configure their product line, Nadi et al. [124, 125] proposed a static analysis approach to extract software configuration constraints from C code. Their approach uses lines that throw errors (via the #error macro) and determines under which configurations these error lines are developed and can be executed. They evaluated their approach on four highly configurable software systems, and found that their approach is highly accurate.

Kenner et al. [102] proposed an approach to help developers detect type errors in a product line's variants. Their approach starts by simplifying the source code, which is then parsed to

an abstract syntax tree, from which their approach builds a set of constraints under which types need to be checked. These constraints are then solved to find which variant has a type error.

Medeiros et al. [117] compared 10 sampling algorithms used to test product line variability, such as pair-wise algorithms (testing a default configuration, then changing a pair of options together for each additional test), and one enabled (testing a baseline configuration, then enabling one option at a time for each additional test until all options have been enabled). Their evaluation consists of finding a tradeoff between the effort in terms of number of samples required to test a highly configurable software and the number of faults an algorithm is able to detect. While algorithms with larger sample sizes are able to detect more errors, many algorithms were found that provide a good balance between sample size and amount of detected errors.

To help developers fix their product line configuration errors, Xiong et al. [210, 211] proposed and evaluated an approach that generates fixes to configuration errors. These fixes do not only propose which options are misconfigured, they also propose a range of possible and correct values that these options should respect. Their approach relies on existing software configuration constraints.

Since this thesis focuses on run-time, i.e., non compile-time configuration options, our findings could not be generalized to conditional compilation-based configuration options, as they are managed by two different categories of customers. While compile-time configurations options are generally modified by developers, run-time configuration options are managed by operators, sysadmins, and end-users, i.e., a combination of technical or non-technical users. However, we think that it will be interesting to replicate the approaches evaluated for compile-time configuration options on run-time configuration options, and vice-versa. For example, one can evaluate the 10 sampling algorithms discussed by Medeiros et al. [117] on run-time configuration options.

CHAPTER 3 RESEARCH PROCESS AND ORGANIZATION OF THE THESIS

We present in this Chapter our research methodology and the structure of this dissertation. The goal of this thesis is to improve the quality of configurable software systems, which is why we first need to define and understand the process of engineering configuration options, and which related challenges developers face. As a second step, we then propose and evaluate approaches to address major technical challenges encountered during typical configuration engineering activities, and to help debug a complex type of configuration errors. As presented in our research hypothesis, this thesis has three main parts:

- The first part of our thesis consists of understanding how developers manage software configuration, which configuration challenges they deal with, and what practitioners and researchers can propose as recommendations to guarantee a good software configuration quality. This part is addressed by Chapter 4 and 5.
- The second part of our thesis consists of empirically evaluating the impact of source code best practices on the quality of software configurations. With the aim of addressing major technical challenges identified in Chapter 4, Chapter 6 addresses this part of our thesis.
- The third part of our thesis consists of evaluating the impact of source code analysis techniques on debugging configuration errors in the specific context of multi-layer architectures, which is discussed in Chapter 7 and 8.

3.1 Software Configuration Engineering Process

While configuration options introduce significant flexibility to a software system, supporting a large number of configuration options on the other hand makes a software system more complex to customize, handle, and understand. That, in turn, has a negative impact on the correctness of a software system. Failures induced by configuration errors represent one of the most challenging and prevalent problems in software engineering [215].

To address the challenges that are caused by software configuration, many studies have been conducted in the research literature, yet most of them focus on debugging configuration failures, testing configuration options, or finding an optimal configuration. However, many other activities might exist and be performed by practitioners, together forming a process of

configuration engineering. Practitioners might also face different challenges on each of these activities, while experts can have a set of recommendations to suggest based on their own experiences.

3.1.1 Investigating the Process of Configuration Engineering, its Challenges, and Expert Recommendations

To identify the activities that practitioners follow on the process of configuration engineering, understand which challenges practitioners face on each of these activities, and what suggestions experts can recommend in order to improve the quality of software configuration options, we conducted a qualitative study in Chapter 4 in which we address the first part of our research hypothesis.

We started our qualitative study by conducting 14 semi-structured interviews to better understand the process of creation and maintenance of configuration options. We recorded and analyzed these interviews via a card sort technique, via which we identified an initial set of activities, challenges, and recommendations on software configuration engineering. We then enhanced this set via a survey of 25 closed and 5 opened questions. At the end of this survey, we received 229 responses that we analyzed quantitatively and qualitatively.

Our interviews and survey were followed up by a systematic literature review, with the aim of finding which activities of the configuration engineering process are less covered in the literature, and what guidelines, techniques, and approaches researchers are recommending to practitioners. We classified the 106 papers related to configuration options that we found using a card sort technique.

In the end, we found that the process of configuration engineering constituted 9 activities, in which practitioners face 22 different challenges, while experts and our systematic literature review come up with 24 recommendations. These challenges can have a negative impact on the quality of a software system in terms of maintainability, usability, understandability, and correctness. Our study also showed that multiple configuration engineering activities are not covered well in the literature. In fact, this opens new research opportunities for future work.

3.1.2 Investigating the Usage of Frameworks Dedicated to Software Configuration

One of the things that we learned in Chapter 4 is that practitioners mostly use basic IO libraries to read configuration options from configuration files, while many sophisticated frameworks dedicated to configuration options exist in practice. On the other hand, read-

ing options via a simple IO library makes configuration options harder to comprehend and maintain. Hence, it is necessary to study the popularity and usage of existing configuration frameworks in open source projects to provide guidelines on how to select a suitable configuration framework.

To achieve these objectives, Chapter 5 conducts an empirical study in which we considered 11 configuration frameworks and their usage in around 2,000 Java projects. Through our analysis of the 11 configuration frameworks, we proposed a taxonomy of features one can consider to select a suitable configuration framework, while via our study of their prevalence in open source projects, we surprisingly found that the most basic configuration frameworks are the most popular.

We also studied which of the taxonomy features are the most relevant by quantitatively analyzing their impact on the effort required to maintain software configuration options. For this goal, we built a logistic regression model that gives indications on which metrics are relevant indicators of maintenance effort. Examples of these metrics are the quality of documentation, the maturity, and stability of a framework.

3.2 Analyzing the Impact of Best Practices on the Quality of Software Configuration

Chapter 6 addresses major technical challenges that we identified in Chapter 4, in the context of our second research hypothesis. We learned in our initial interviews (Chapter 4) that one should consider configuration as code, and consequently apply source code best practices to configuration options as well. We iteratively prototyped these ideas in a concrete framework that defines the metadata of configuration options inside the source code, then automatically generates configuration files. During each of the following interviews, we received new feedback about the prototype and its underlying ideas, which we then refined. The resulting framework, which is called Config2Code, incorporates the following principles:

- The first principle consists of considering run-time configuration as source code, allowing developers to benefit from source code best practices for their software's configuration as well.
- The second principle consists of using the principle of encapsulation on configuration option access.

- The third principle consists of automating the generation of configuration files to synchronize configuration files with those options that are actually being used in the source code.
- The last principle consists of automatically validating that options and their values respect predefined constraints.

We evaluated these principles via a user study, whose goal is to compare the ability of these principles to address the technical challenges of configuration engineering identified in Chapter 4. We compared Config2Code against the configuration framework Preferences, which is a popular Java framework that is not respecting the 4 principles.

To perform our user study, we prepared two versions of the open source project JabRef, which is a highly configurable open source application. Since the existing version is implemented with Preferences, we modified the source code of JabRef to use Config2Code.

The study incorporated 11 tasks based on the 9 configuration engineering activities that we found in our first qualitative study (Chapter 4). These tasks range from the creation of new options and maintenance of options to the debugging of configuration failures and reviewing of a patch that introduces a new option. During the user study, 55 participants had to answer questions on paper, while we recorded their tasks and activities on screen. They also filled out at a survey, in which they express their observations and comments.

We analyzed this data by focusing on the tasks' correctness and time required for each participant to finish each task. We found that the four principles help practitioners ensure correctness or minimize the time to achieve 72% of our user study tasks.

3.3 Cross-stack Configuration Errors

Many approaches were proposed to help practitioners debug configuration failures (one of the major configuration engineering activities identified in Chapter 4), yet none of them focused on cross-stack configuration failures. These are errors that occur due to a misconfigured option in one or multiple layers of a stack, e.g. the LAMP stack, in which each layer is consuming services from lower ones, possibly developed in a different programming language.

To address our third hypothesis, we started by studying the extent to which cross-stack configuration errors exist, by analyzing the impact of a configuration option of one layer on another layer. Then, we studied the prevalence and impact of cross-stack configuration errors in real cases, before proposing and evaluating an approach on debugging this type of configuration errors.

3.3.1 Understanding the Impact of an Option on other Layers

In Chapter 7, we studied the potential of cross-stack configuration errors. The goal of this study is to understand how likely one can get a cross-stack configuration error. In this study, we analyzed configuration options of the top three layers of the LAMP stack, which are the PHP-interpreter, web-app platform such as Wordpress, and that platform's plugins.

In this study, we focused on Wordpress and its plugins as a case study of the web-app platform and its plugins layer. Before studying the impact of one layer's options on other layers, we first need to understand how Wordpress and its plugins use configuration. This is why we analyzed the source code of Wordpress and 15 of its plugins, and found which mechanisms they are using to manage their software configuration.

The increasing number of configuration options can also be a good indicator about the potential of cross-stack configuration errors. That is why we also analyzed the evolution of WordPress and 15 of its plugin configuration options across different versions.

We then quantitatively analyzed the source code of 484 Wordpress plugins to find how many Wordpress and PHP-interpreter options each plugin is using, and we analyzed the WordPress source code to find how many PHP-interpreter options it is using. More lower layer options used by a given layer indicates a large potential for cross-stack configuration errors.

Finally, we analyzed the whole set of 484 plugins to find, for each option, how many plugins are using it at the same time. That indicates the impact of changing one single option on the set of installed plugins.

3.3.2 Understanding Real Cases of Cross-stack Configuration Errors, and Proposing an Approach to Debug such Errors

In Chapter 8, we aim at (1) studying real cases of cross-stack configuration errors, and comparing them to single-layer ones, and (2) proposing and evaluating an approach that is based on source code analysis techniques. The goal of this study is to address the third hypothesis, in which we hypothesize that source code analysis techniques can help debug configuration errors, not only within one single layer as confirmed by the research literature, but also across a stack of layers.

To achieve the first objective of this study, we conducted a qualitative analysis of 1,042 StackExchange discussions related to software configuration. Via a card sort approach, we analyzed different characteristics of these errors, such as their impact, in which environment they occur, and how many options need to be changed to fix each error. We also collected some quantitative metrics like the time required to fix such errors and the number of people

involved in this discussion, which can be an indicator about the difficulties of fixing cross-stack configuration errors.

After confirming the severe impact of cross-stack configuration errors, we proposed a modular approach to debug such errors. It recommends a set of configuration options that are most likely to be misconfigured. This approach is modular, as it allows to use and compose existing source code analysis techniques to find which option is misconfigured.

We evaluated the impact of this approach on 36 real configuration errors, which we found in our initial analysis of stackexchange conversations. This evaluation of our approach considers the accuracy of its recommended options, and time required to find the culprit options.

CHAPTER 4 ARTICLE 1: SOFTWARE CONFIGURATION ENGINEERING IN PRACTICE - INTERVIEWS, SURVEY, AND SYSTEMATIC LITERATURE REVIEW

Mohammed Sayagh, Noureddine Kerzazi, Bram Adams, and Fabio Petrillo
Submitted to the IEEE Transactions on Software Engineering (TSE)

Abstract: Modern software applications are adapted to different situations (e.g., memory limits, enabling/disabling features, database credentials) by changing the values of configuration options, without any source code modifications. According to several studies, this flexibility is expensive as configuration failures represent one of the most common types of software failures. They are also hard to debug and resolve as they require a lot of effort to detect which options are misconfigured among a large number of configuration options and values, while comprehension of the code also is hampered by sprinkling conditional checks of the values of configuration options. Although researchers have proposed various approaches to help debug or prevent configuration failures, especially from the end users' perspective, this paper takes a step back to understand the process required by practitioners to engineer the run-time configuration options in their source code, the challenges they experience as well as best practices that they have or could adopt.

By interviewing 14 software engineering experts, followed by a large survey on 229 Java software engineers, we identified 9 major activities related to configuration engineering, 22 challenges faced by developers, and 24 expert recommendations to improve software configuration quality. We complemented this study by a systematic literature review to enrich the experts' recommendations, and to identify possible solutions discussed and evaluated by the research community for the developers' problems and challenges. We find that developers face a variety of challenges for all nine configuration engineering activities, starting from the creation of options, which generally is not planned beforehand and increases the complexity of a software system, to the non-trivial comprehension and debugging of configurations, and ending with the risky maintenance of configuration options, since developers avoid touching and changing configuration options in a mature system. We also find that researchers thus far focus primarily on testing and debugging configuration failures, leaving a large range of opportunities for future work.

4.1 Introduction

In September 2010, Facebook users experienced the following severe outage [96]:

> "*Early today Facebook was down or* ***unreachable*** *for many of you for approximately* ***2.5 hours****. This is the worst outage we've had in over four years. The key flaw that caused this outage to be so severe was an unfortunate handling of an error condition.* ***An automated system for verifying configuration values ended up causing much more damage than it fixed****.*
> ...
> *We're exploring* ***new designs for this configuration system*** *following* ***design patterns*** *of other systems at Facebook that deal more gracefully with feedback loops and transient spikes."*

The above issue was not (directly) caused by a bug in the source code, but rather by the complexities of dealing with configuration in a software system. Instead of hardcoding the values of deployment-dependent constants such as URLs, or the choice of algorithm or feature that needs to be used, software systems extract such values into configuration options, which can then be modified by operators without having to re-compile the source code. As such, software systems can easily be ported to a different environment or user base simply by reconfiguring the options. In the above example, Facebook used a database for storing these values.

While configuration options provide substantial flexibility to developers and users, they create a huge space of variability that can be hard to maintain [95, 185, 215, 219]. Configurable software systems can have hundreds of configuration options [212], which gives an effectively inexhaustible number of variants (2^{100} variants for a software system with 100 boolean options). Since it is impossible to test all variants before release, and options are not necessarily limited to only 2 values (they could be arbitrary strings!), certain configurations could lead to invalid behaviour of a software system or even an inability to boot up. Such situations are referred to as "configuration failures".

Configuration failures not only have a huge impact on availability, but they are expensive and common in both open source and commercial software systems [215]. Yin et al. [215] showed that configuration failures can account for 27% of all customer-support cases in industrial contexts, while a well-known Google engineer [199] highlighted them as one of the most important future research directions. In addition, other researchers [43, 95, 185] focused on preventing configuration failures, where a second string of efforts, including [45–47, 68, 69,

136, 147, 192, 204, 217–219, 224], have been conducted to help users troubleshoot such kinds of problems.

Despite all this research on dealing with configuration failures, especially from the end users' perspective, there is hardly any work on understanding the effort involved in "configuration engineering" required from practitioners, i.e., the development activities that make up the process to establish and maintain configuration options throughout the lifetime of a software project. In general, there is a lack of understanding of the challenges practitioners experience with configuration engineering as well as best practices that are recommended. Such an understanding would not only open up new research directions (other than targeting configuration failures), but also document a catalogue of configuration engineering challenges and workarounds. The latter catalogue is essential to developers[1], since it is hard for them to determine this knowledge only from their own system's context.

Hence, this paper investigates the state of the practice and challenges of configuration engineering in the trenches by conducting a set of interviews, a large Java practitioner survey and a systematic literature survey. These allow us to make the following contributions:

- Deriving the typical process of configuration engineering, comprising 9 essential activities carried out by developers to create and maintain configuration options in a software system. This process starts by the creation of new configuration options, followed by the management of storage medium and data format (type) of these options and the design of how to access these options in the source code. The process also includes the comprehension and maintenance of options, sharing of knowledge about them, and both prevention and resolution of configuration failures.

- Identifying the configuration engineering challenges faced by practitioners related to the 9 activities. For instance, we found that development teams typically do not plan the design of potential configuration options nor do they have an adequate process for reviewing code changes involving options, which might have a negative impact on the quality, understandability and maintainability of software products. Similarly, perhaps surprisingly, developers are afraid of removing options from their code base in order not to break existing functionality.

- Collecting a set of recommendations, both from practitioners and researchers, to address the challenges. The recommendations include basic guidelines on how to organize options, when developers should decide adding a new option, what should be in configuration documentation, and where developers should define it. They also include a set

[1]This was mentioned a number of times in our developer survey.

of guidelines on how to access configuration in the source code, and how to debug and resolve configuration failures. A common thread within the guidelines is for developers to consider configuration options as code. Similar to source code, they should review new options, define and respect a clear naming convention, encapsulate access to configuration options within dedicated helper classes, and log used configuration values to help debug configuration failures. In turn, the resolution of these failures should be documented.

- A public dataset of our analysis to the interviews [19], and survey responses [31].

Given the large scope of software configurability, we focus explicitly on software configuration options that do not require re-compilation nor re-deployment to take into account new values, i.e., options whose value can still be changed at run-time. In other words, we focus on post-deployment configurability, and leave other forms of configuration engineering as future work.

The remainder of the paper is organized as follows: Section 4.2 presents the background about software configuration. Section 4.3 presents our study methodology. Section 4.4 presents the 9 identified configuration engineering activities, while Section 4.5 presents the challenges that we identified from qualitative and quantitative analysis of the 14 interviews and the 229 survey responses. Section 4.6 provides the recommendations identified during our interviews, survey and systematic literature review. Section 4.8 discusses threats to validity, while Section 8.7 concludes with insights and future work.

4.2 Background on Software Configuration

This section defines the notion of software configuration used by this paper and discusses different types of configuration.

4.2.1 Software Configuration Options

Configuring a software system consists of customizing and adapting an application as well as its execution environment [88] to different users' requirements and contexts [172]. Conceptually, a configuration option of a software application corresponds to a key-value pair, for example to quickly enable or disable features and algorithms. A key represents a configuration option's name, whereas its value represents the choice by a practitioner or user regarding that option for a specific instance of the system. For example, to disable file uploads in a Wordpress installation, one can just switch off the PHP configuration option

"file_uploads", without the need to change the actual PHP source code. "file_uploads" and "Off" are respectively the key and value of the corresponding configuration option.

One of the most common storage media for configuration options are ordinary files, which we refer to by "configuration files". Developers typically can choose to store their software system's options in different textual file formats, such as "txt", "ini", "xml", or "json", or even to use or design a configuration domain-specific language (DSL) [51] for their software system. Apart from configuration files, developers could also store their configuration options in a relational database, which can be accessed via a user interface, or even directly modified by end users. More recently, distributed key-value stores like Apache ZooKeeper [7] have become widely used for managing configuration options in distributed environments. Alternative means of storing configuration options would be as conditional compilation constants or even as command line arguments passed to an application.

4.2.2 Roles Involved in Software Configuration

Many roles are involved in the creation and usage of software configuration, which we can classify in two main categories: (1) the creators and (2) the consumers of a software configuration. The creator category consists of a variety of engineering roles, including architects, designers, developers, and testers. All of these are technically minded, and have access or influence on the resulting source code of a software system.

On the other hand, the consumer category comprises both technical and non-technical configuration users. Technical users include the same engineering roles as the creator category, but also roles responsible for software operations, such as deployers, integrators, and operators. These users usually manipulate technical configuration options related to both the execution environment and application, such as resource paths, accounts or options to tweak the security and performance of an application.

In contrast, non-technical users are the final consumers of a software application and do not necessarily have a technical background. They mostly change configuration options to help them achieve their objectives, such as information about their account, preferred algorithms or features, or GUI configuration.

4.2.3 Configuration vs. Binding Time

Given that configuration options could target different roles, who have different types of access to the source code and/or deployed version of a software system, software configuration typically is performed during three different phases. These phases determine the "binding

time" of configuration options [116], i.e., the moment on which the options are assigned their values: before deployment, during deployment or after deployment.

Pre-deployment options typically are implemented using compile-time configuration options, which decide whether particular code snippets will be compiled in by the build system. For instance, in C/C++, a code snippet occurring between "#ifdef Config_Option" and "#endif" delimiters will be observed and hence compiled by a compiler. This happens only if the "Config_Option" option has been defined in a source code file or as a command-line parameter for the compiler (typically specified within a build script), otherwise the corresponding code simply is unavailable to end users in the resulting executable. Changes to compile-time options require recompilation of the source code.

An alternative means to implement pre-deployment binding of configuration options is to use build scripts such as Makefiles, which can select the set of source code files that should be compiled, or even the set of 3rd party libraries (and their specific versions) that should be linked to the compiled application code. This is common in large systems such as the Linux kernel, where a kernel should be as streamlined as possible and hence only contain the drivers expected on the target platform.

Next, during deployment (sometimes called "installation"), modern web apps use domain-specific languages (DSLs) such as Puppet, Chef, or Ansible to install and configure the environment of an application, i.e., the virtual machines, containers, or servers on which the application will run, with the required operating system, database, web server, selection of 3rd party libraries, network connections, etc. Such DSLs are not used for desktop or mobile apps, where instead an installer performs any environment-related configuration. Configuration scripts written in one of these DSLs or bundled within an installer basically auto-detect values for environment-related options, use those values to fill out (bind) template configuration files (e.g., of a web server or even the application), then move the instantiated templates to the right location in a virtual machine, container or computer.

In contrast to pre-deployment binding, however, options bound during deployment can still be tweaked after deployment. This is because each component in the environment of an application A is itself a software application, with its own collection of pre-deployment, deployment-time and post-deployment options. The main difference, from A's perspective, is that most of its environment's options will be bound during deployment, while most of A's options will be bound afterwards, because the latter are expected to be handled by the end user.

Finally, after deployment, two different binding times are possible, i.e., load-time and execution-time. Load-time configuration options are read by the source code when a software system

boots up. Upon boot, the software system reads all the options that are defined in its configuration storage medium. Changes to these options just require restarting the application. Execution-time configuration options are the most flexible form of configuration options, since they do not depend on recompilation nor rebooting of the whole application. Instead, user modifications can be taken into consideration immediately after changing, provided the method responsible for configuration access is re-executed.

4.2.4 Run-time Configuration Options

This study focuses on the process followed and challenges experienced by developers when building in software configuration aimed at end users, sysadmins and operators. These roles have access to a system during and/or after deployment, hence at first sight our focus is on options bound during those two phases, i.e., options that are propagated via and can impact data and control flows in the source code. Pre-deployment options fall outside our scope, since such options typically only filter out unnecessary parts of the source code, then "disappear" during compilation.

However, the distinction between the binding times presented in the previous section is not always as strict, hence defining the scope of our study in terms of them introduces ambiguity. For example, during compilation a configuration tool like autoconf might have bound a system-specific value (e.g., the maximum integer size) to a C variable representing a configuration option, while this variable might be re-bound to a different value by end users at run-time. Similarly, most of the configuration options bound to an auto-detected value during deployment can still be overridden at run-time. For example, during installation an installer could detect 4 GB RAM on a laptop, yet after upgrading the memory to 8 GB RAM, a laptop owner should not need to reinstall (redeploy) the application, but just manually change 4 to 8 in the application's configuration file. In these (and other) cases, the exact type of binding is unclear.

To deal with this ambiguity, we instead define the scope of our study in terms **"run-time configuration options", i.e., configuration options whose value can still be changed by the end user, without having to re-deploy**. This is an approximation that roughly coincides with deployment-time and post-deployment options, while still including edge cases like the examples of maximum integer size and RAM amount. On the other hand, most of the pre-deployment options (in particular conditional compilation-based options) are excluded. Due to their different target audience, we believe that they likely face different challenges than run-time options, and hence different workarounds and best practices. Future work should replicate our study on pre-deployment configuration options.

4.2.5 Configuration Failures and Faults

Assigning an incorrect value to a configuration option can lead the software to behave incorrectly, which is referred to by the term "configuration failure". Such failures can have different manifestations, including error messages, performance degradations, hidden and silent execution failures, or even crashes or refusal to boot up. If they occur in production, they can incur serious financial losses [4, 12].

Typical causes for configuration failures, i.e., configuration faults, include typos (e.g., "memory_limit = 64" instead of "memory_limit = 64M") as well as unrealistic or plain wrong values. A second type of configuration faults correspond to options whose value is syntactically correct, but does not match with the execution requirements, (e.g., "max_execution = 60" when a time-consuming script that requires more than 60 seconds is executed). For other types of configuration faults, including inconsistencies between configuration option values, we refer elsewhere [215].

4.2.6 Software Configuration Engineering

We define software configuration engineering as the discipline that encompasses activities involved in the creation, integration, and maintenance of run-time configuration options in a software application. For example, adding a new option to an existing code base, and managing configuration failures are two common configuration engineering activities. Other engineering activities also apply to software configuration including documentation and code review. While there exist substantial work on several activities related to software configuration engineering, many others are not covered at all by literature. Furthermore, there is no study that focuses on the full lifecycle of software configuration options. In fact, until now no official name existed for the overarching discipline of software configuration engineering. Hence, the next section discusses the methodology of our exploratory study on the different configuration engineering activities (as well as their challenges and recommendations).

4.3 Study Methodology

In order to understand the activities making up the discipline of software configuration engineering, the challenges involved with those and best practices that exist, we used a mixed-methods approach involving semi-structured interviews, a survey with open source stakeholders and a systematic literature survey. The 14 interviews with experts allowed to define 9 essential configuration engineering activities as well as to identify an initial list of challenges of these activities. The activities and challenges were used as inspiration for the open-ended

and closed questions of our survey. The 229 survey responses allowed to validate the activities and challenges, as well as to identify best practices used to deal with these challenges. Finally, to put our findings into perspective, we performed a systematic literature survey focused on the 9 activities. The rest of this section provides more details about our study methodology.

4.3.1 Semi-structured Interviews

Since we aim to understand activities, challenges and best practices used in practice, and no explicit configuration engineering process could be used as reference, we opted to start with semi-structured interviews of experts. Then, to put the experts' input into context, we performed a larger-scale survey building on the experts' answers. To conduct the semi-structured interviews, we first had to define the base set of open questions to discuss with experts, which only requires high-level guiding questions to explore concrete experiences and insights of the interviews. These open questions were obtained through brainstorming of the paper's authors around the central theme "*How do practitioners design, develop and maintain the configuration system of their application?*" To avoid confusion during the interviews (and later survey), we clearly defined the focus of our study in terms of run-time configuration options.

Throughout the brainstorming, we leveraged the extensive industrial experience with configuration of two of the authors, as well as the results of our earlier work on software configurations [162–164]. To mitigate the risk of missing major questions or themes, we chose a semi-structured format for the interviews. Such a format helped us complete the base themes and questions by allowing interviewees to change the direction of the interview if interesting new topics are being covered. Such topics could then be integrated in our base interview guide for future interviews. The final interview guide is available online [27].

The 14 interviewees were recruited from different companies and have different software engineering roles, ranging from developers to software architects and managers, including one infrastructure (i.e., environment) architect. Each of these interviewees belongs to a different company, except for two interviewed experts who did work in the same company, but under different roles (manager and integrator). The 13 companies also cover a wide range of domains, from governments to banking systems and online retailers. As presented in Table 4.1, the 14 interviewees had between 5 and 28 years of software engineering experience, with only four people having less than 10 years of experience.

Each interview was performed by two of the authors, one of which was the interviewer and one of which the scribe taking notes. The interview would start from an open question,

Table 4.1 Interviewed subjects' experience and role.

Subject	Experience (#years)	Role
P1	28	Developer and Manager
P2	27	Researcher
P3	21	Developer
P4	15	Manager
P5	14	Developer and Industrial Researcher
P6	14	Infrastructure Architect
P7	13	Manager
P8	12	Developer and Architect
P9	10	Developer and Architect
P10	10	Developer
P11	8	Integrator
P12	7	Developer
P13	6	Developer
P14	5	Developer, Architect, and Manager

then the interviewee's answers would help select the next question to move to. Although we made sure that a typical interview could be finished within one hour, the semi-structured interviews eventually had a length of one to three hours.

4.3.2 Card Sort Analysis of Interview Data

In the second step, two human raters (authors) qualitatively analyzed the interview results by following a card sort approach [155]. Card sorting is a categorization technique that is widely used in information processing to derive taxonomies from input data [155]. In our case, the two raters printed the interview transcripts and notes, then cut those up in snippets containing relevant information. These snippets were then classified collaboratively into clusters that represent a common topic, for example "testing configuration options". Once finished, the raters archived the classified snippets for two days before doing a second iteration of classification in which topics related to the same high-level configuration-related activity (e.g., "quality assurance") were clustered together.

The resulting clusters enabled us to identify 9 configuration-related activities, together with challenges and anecdotal evidence related to the activities. The other two authors then validated these results. The final results of the card sort are available online [19].

4.3.3 Survey

In order to extend our understanding of the configuration-related activities and challenges identified via the interviews, and to identify how representative they are, we conducted a large 20 minute survey containing 25 closed and 5 open-ended questions. To arrive at this final list of questions, we first identified 56 questions from the individual topics (card sort clusters) within the 9 obtained configuration activities. Two of the authors discussed and reduced this list to 46 questions, after which the two other authors took the survey to evaluate its clarity, duration and relevance to our study. After addressing the identified issues, we sent the survey to personal contacts (including a number of open source developers). The 9 answers that we received were then used to validate and improve our questionnaire. We re-organized the order of questions into 4 pages (instead of 12), revised two unclear questions and could further reduce the number of questions to 30. Our survey questionnaire is available online [32].

We then sent out 2,000 personalized emails inviting open source project developers to participate in our survey. These developers were selected from the 1,000 Java projects on GitHub with the most commits, as obtained from the GHTorrent [78] database. This selection strategy allowed us to focus on mature and active projects, ignoring Github repositories that are not software projects (e.g., repositories for courses or student exercises). We focused on Java projects to control for the impact of programming language on configuration options, and since it is the most popular programming language on the TIOBE index at the time of performing this study. For each selected repository, we selected a sample of contributors, whose email address we obtained via the Github API [18]. We manually checked our targeted developers before sending emails.

Eventually, we received 229 responses, yielding a response ratio of 11.5%, which is a reasonable percentage compared to surveys in other domains [29]. Note that our invitation email explicitly asked the surveyees to answer the survey from the point of view of the specific project for which we had contacted them. Furthermore, the 229 responses do not include the 9 responses used for validation of our questionnaire.

The work experience of the 229 survey participants varied from a minimum of 1 year to a maximum of 45 years, with a median of 11 years and average of 13.35 years. The majority of these participants are developers (184), while 82 are architects, 27 managers and 19 academics (students, professors or lecturers). Note that some participants had more than one role at their organization. 3% of the participants have contributed to projects with a huge number of configuration options (between 1,000 and 10,000), 20% to projects with a large number of configuration options (between 100 and 1,000), and 51% to projects with a median number of options (between 10 and 100). Only 26% of the participants had contributed to projects

with less than 10 options, indicating that the majority of survey participants are indeed practitioners who have been exposed to software configuration and hence can offer a large variety of perspectives.

4.3.4 Card Sort Analysis of Survey Data

Closed survey questions were analyzed quantitatively, while open-ended questions were again analyzed using card sort analysis, similar to the interview data. Given the scale of the obtained survey data, we performed its card sort electronically, eventually obtaining clusters representing configuration-related challenges and clusters representing best practices to address challenges, each with a variety of sub-challenges or -practices, and anecdotal evidence [8, 9].

4.3.5 Systematic Literature Review

Finally, to identify which best practices in configuration engineering are not yet covered by earlier research as well as to guide practitioners to published approaches and solutions, we performed a "systematic literature review". The **objective of our literature review** is to explore the research literature that focuses on software configuration, then classify it based on the 9 configuration activities, and the challenges and practices identified during the interviews and survey. We follow the systematic literature survey process prescribed by Kitchenham et al. [100].

The approach used in our literature survey consists of obtaining papers studying run-time software configuration from the online Compendex, Inspec, ScienceDirect, and Web of Science databases. We obtained a set of 632 papers, which we obtained by using the following search criteria and queries:

- **Criterion 1**: In order to select papers that discuss configuration, we focused on those in which the title and abstract contain the keyword "config*", "misconfig*", or "misconfig*".
- **Criterion 2**: To eliminate papers that are not related to software engineering, we only retained papers whose venue title contains the keyword "Software" (like International Conference of Software Engineering).
- **Criterion 3**: To reduce the amount of papers to classify in the first iteration, we initially focused on papers published in the last 10 years (From 2007 to 2017). Later on, we used snowballing (see below) to cover papers earlier than 2007.

- **Criterion 4**: Papers should be written in English.

Inclusion and exclusion criteria, we manually studied these 632 papers by classifying them into two categories: (1) papers focusing on run-time software configuration, and (2) papers that are out of scope for our research (e.g., on compile-time software configuration or software product families). For this initial classification, we read each paper's title, abstract, introduction, and conclusion.

At the end of this iteration, we obtained 69 papers that are related to run-time software configuration. Afterwards, 6 more papers were eliminated as out of scope based on peer-review and discussion between authors, resulting in 63 papers.

We then added 7 extra papers that we already knew (bringing our tally to 70), but were missed by the criteria above. 5 out of these 7 were published in venues that do not contain the keyword "Software", while two papers were not in the databases that we selected (Compendex, Inspec, ScienceDirect, and Web of Science).

To cover a larger search space, we then performed **"snowballing"**, i.e., the identification of additional papers from the references of the 70 papers selected in the previous steps, repeated recursively from each of the newly selected papers' references. At the end of this iteration, 36 new papers were added to our dataset, yielding 106 papers related to software configurations.

4.3.6 Card Sort Analysis of the Systematic Literature Review Data

To complement the practitioners' recommendations with the state of the art discussed in academic literature, we classified the final set of 106 papers according to the 9 configuration activities that we identified via the interviews and survey. Each paper could cover more than 1 such activity. The goal of this classification across activities is to lead practitioners to approaches for their specific context, to confirm practitioners' recommendations with existing researchers' validations, and to identify configuration research areas that have been covered less in detail and hence should likely be the focus of future work.

In order to conduct the card sort analysis, we followed these steps:

1. Each author focused on a subset of the 106 papers that we obtained at the end of the snowballing activity.

2. Each author read each of his papers to deeply understand its focus and contributions.

3. Each of the papers was then classified according to one or more of the 9 activities identified during the interviews and survey. At the end of this step, each paper was either classified into zero, one or more categories based on its contributions.

4. After the classification, each of the authors reviewed the papers of another author to verify the classification. If two people disagreed about a classification, they together discussed it to arrive at a final decision.

5. Within each activity, we then matched papers with the corresponding challenges that they are addressing.

The next three sections present the card sort results regarding configuration engineering activities, challenges and best practices, each supported by quantitative and qualitative evidence.

4.4 Configuration Engineering Process

The first goal of this study is to identify the configuration engineering process used in a typical software project. This process basically consists of a set of activities performed routinely by different software engineering actors, including developers, architects, and managers. These activities were identified from the card sort analysis of the interviews, and were later confirmed by the survey (and implicitly by the literature survey). The resulting list of 9 activities (visualized in Figure 4.1) should be taken into account by any new developer or software project when estimating development or maintenance effort. We also hope that these activities will become the focus of future research by our community to help address the challenges presented in the next section.

Table 4.2 shows the mapping between the 9 activities to the survey questions that covered them. Note that the last two survey questions were open-ended, allowing the surveyed developers to express challenges and recommendations on any configuration activity.

A1.Design and Creation of Configuration Options The most obvious activity, which 92% of survey participants performed at least once, is to add a new configuration option to a software system. This involves choosing a name, a type (e.g., boolean or string), a default value, the physical location (storage medium) to store the option's value, and providing comments or other documentation. Most of the participants create a new option to either configure technical aspects (like the memory usage or database credentials) of an application or to configure the actual functionality of the application.

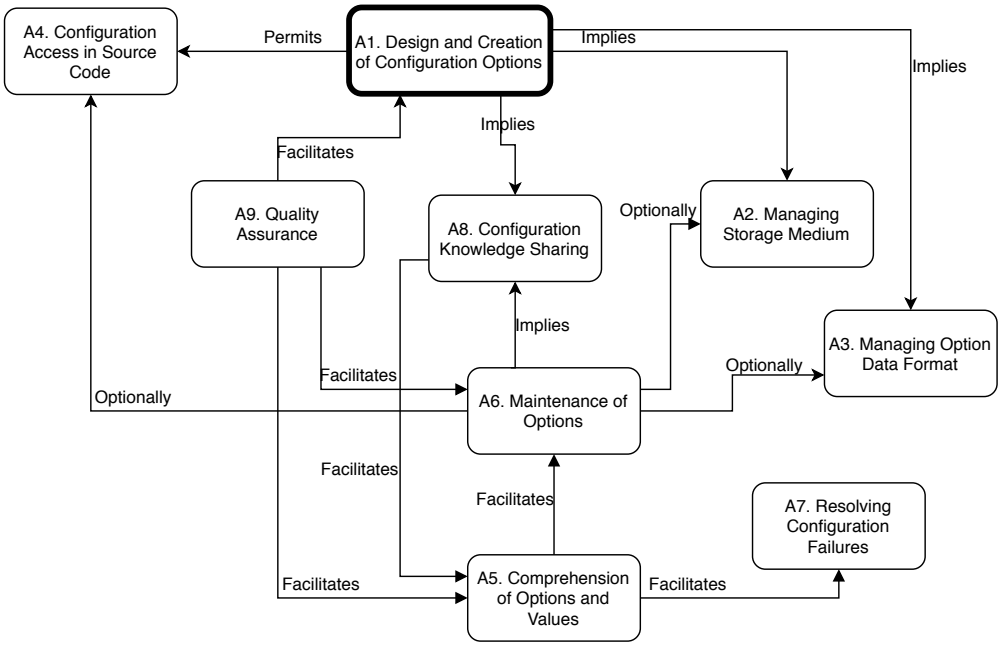


Figure 4.1 The process of software configuration engineering.

These options can be specified in the user requirements or architecture document, or can be "invented" by an architect or developer, for example to "externalize" a source code constant into a configurable option. Such externalization refers to the act of allowing the value of an option to be changed from outside the source code, hence avoiding recompilation.

A2.Managing Storage Medium The storage medium for configuration options, i.e., the physical location where options and their values are stored (and can be changed), is mostly determined by personal preference as well as by the intended "binding time" of an option. The

Table 4.2 Mapping of each survey question (in the Appendix) to the activity it addresses.

Activity	Question
A1	9, 10, 11, 12
A2	5
A3	12
A4	6, 7, 8
A5	19, 20, 23, 24, 25
A6	21, 22
A7	15, 16, 17, 18
A8	13
A9	14, 26, 27, 28

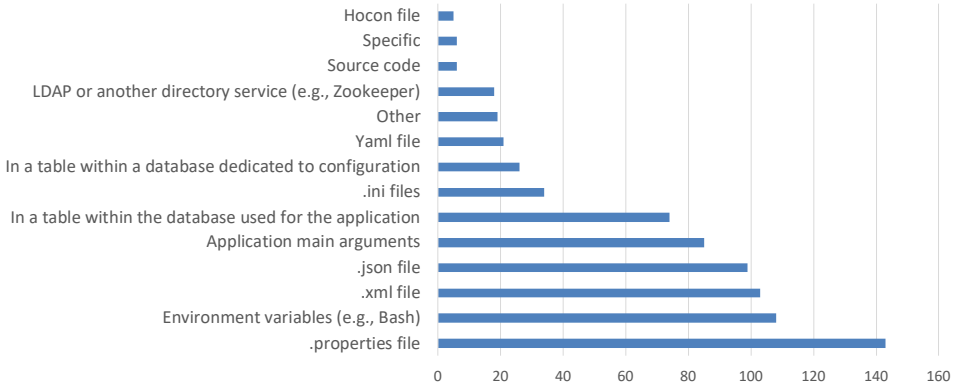


Figure 4.2 Popularity of configuration storage media (based on survey question 5).

binding time is the moment at which a configuration option gets its value assigned (bound) to it, as discussed in Section 4.2.3. For example, options stored as conditional compilation macros or as hard-coded source code variables are bound at compile-time (outside the scope of this study), while most configuration files and program arguments are bound at load-time. To obtain execution-time binding, options are typically stored in databases or (less often) in files. Sometimes more esoteric mechanisms are used, such as checking for the presence of a file to set a boolean option to true. Such a mechanism enables file system permissions to restrict access to an option.

Execution-time binding is used for options whose value could change dynamically (typically functional options), while load-time options are used for options that do not need to change after starting up an application, but do need to be changed by the end-user, like the amount of memory allocated to an application or the credentials for accessing a database. We found that 98% of the surveyed participants use load-time configuration options and 35% use execution-time options. Given the scope of our study (and our focus on Java projects for the survey), no compile-time option usage was recorded.

As shown in Figure 4.2 (based on question 5 of the survey), the top three most popular configuration storage media in our survey are *"properties"* files, *"environment variables"*, and *"XML"* files, respectively enticing 62% (143/229), 47%, and 45% of the surveyed participants.

The reason for listing the management of configuration storage medium of a software application as a separate activity is that it is not just a choice made during creation of an option (A1), but also involves maintaining or even changing the medium over time, for example by splitting up or merging configuration files, renaming them or even moving from files to databases.

A3.Managing Option Data Format A configuration system's data format (or schema) corresponds to the types of values allowed for each option, together with any constraints enforced on these values. Option types can range from pure strings (even to represent numbers or more complex structured data), to simple types distinguishing between string and numeric types, or complex types allowing more advanced, custom data formats.

A popular form of complex configuration format are hierarchical options, which use a nested data structure to store configuration values. Instead of having 5 separate options related to the same feature spread cross the configuration files, nested data structures allow to aggregate these options in a single, composite configuration option. In the survey, we noticed a more recent evolution from purely hierarchical option values towards graph-based structures.

Although the choice of data format is somewhat influenced by the choice of storage medium (a properties file cannot easily model hierarchical values), most storage media can handle any data format, to some degree. One of the main differences is the ability of a storage medium to automatically enforce constraints on the values. For example, XML schemas are able to check that an XML configuration file is valid, while properties files do not have such built-in support.

Similar to managing the configuration storage medium, managing the configuration data format covers more than just the initial choice of format, such as refactoring to another format as an application's features evolve, as well as refinement of configuration data constraints.

A4.Configuration Access in Source Code Once an option has been added, developers should access its value from within the source code in order to enable/disable features, to create connections or to perform specific calculations. To enable such access to the configuration options stored in the storage media, we identified three sets of approaches in use: using a third party configuration API (frameworks such as Java Preferences or Apache Commons), developing a custom API, or a hybrid approach with a custom API leveraging a third party API.

A5.Comprehension of Options and Values To use, manipulate, or maintain an option in the source code, end users and developers first need to understand the goal of that option and its possible values. The surveyed participants use a variety of artefacts to understand an option, ranging from comments in a configuration file, documentation of the source code, external documents, to readme files and manuals. Developers also rely on informal approaches to understand configuration by asking and discussing with their colleagues.

A6.Maintenance of Options Once an option is created, accessed and used in the source code, developers need to monitor its usage and usefulness over time, basically maintaining the

option and the code that is using it. Maintenance activities can vary from renaming options and refactoring the code regions accessing them to removing redundant options. Although our interviewed and surveyed participants agree that maintenance of configuration options is important, they unanimously confirmed that it is not an easy task, as we discuss later. In fact, they consider configuration option maintenance to be the major area where tool support is lacking.

A7.Resolving Configuration Failures A major activity impacting developers and users of an application is to resolve failures that are caused by misconfigured options, i.e., options that were assigned an incorrect (default) value. This involves reproducing the failure to confirm that it is caused by a misconfiguration (and not a code-related fault), debugging the reproduced failure to identify the faulty configuration option and identifying the right value to fix the misconfiguration.

A8.Configuration Knowledge Sharing Sharing development knowledge between team members is a common activity in software engineering. As part of this knowledge, developers also need to share amongst each other important information regarding configuration options. This information ranges from data on newly added options to illustrations showing the impact of an option on source code regions, or metadata about common configuration failures and faults for tricky options. While activity A5 considers the understanding of an option from a configuration option user's perspective, activity A8 considers such understanding from the developers' perspective (e.g., for evolution and maintenance purposes).

A9.Quality Assurance Finally, similar to regular code development, quality assurance is a major activity for assuring the correctness and integrity of a project's configuration, with the goal of improving software configuration comprehension, improving the resilience of an application to configuration failures, reducing the software configuration's complexity, and improving its maintainability. For example, some projects developed static analysis tools to check the semantic constraints on the values of a configuration option (on top of the basic type checks of the configuration data format) or to test a system in the context of a given configuration. Traditional quality assurance activities like code review or tests equally apply to configuration options, their values and their access within the code.

Relations between Configuration Engineering Activities As shown in Figure 4.1, the configuration engineering process obviously starts by the creation of new configuration options (A1). Creating a new option also implies deciding on a storage medium (A2) and on a data format (A3). These two last activities are separated from the creation of configuration (A1), as they are also impacted by the maintain of configuration options (A6). For exanple,

developers can decide to change an option type from a string to a boolean, or to change the storage medium from a simple file to a more powerful database.

Once an option has been created, developers can start accessing it within the source code (A4). It is also important for the people who added the option to document and communicate the intent and constraints of the option to other developers (A8). This is for example important to facilitate the comprehension of options and future changes (A5).

On the other hand, effective quality assurance practices (A9), such as configuration-aware code review or the specification of naming conventions, can improve and facilitate the creation, maintenance and comprehension of an application's configuration system. The comprehension of options and their values (A5) is of fundamental importance for the maintenance of software configuration options (A6) and for debugging of their failures (A7).

The above relationships were derived during card sorting, in parallel with the identified activities. Most of these relations are rather trivial, while some of them require further scientific validations. For example, it is trivial that creating an option implies managing the option's storage medium. However, it is less clear that quality assurance (A9) facilitates comprehension of options (A5). The survey did not require respondents to evaluate the relationships.

4.5 Configuration Challenges

This section presents the challenges that we identified from card sort analysis of the 14 interviews and the 229 survey responses for each of the 9 activities. Figure 4.3 summarizes these challenges, ordered by the number of survey participants who mentioned them.

Although the interviews and survey address two different population categories (which are respectively industrial and open source Java engineers), we did not observe any inconsistent results between both categories (except C9.1). Therefore, we merged the results and discussions of both categories in the remainder of the paper. However, it might be worth to explore more deeply the differences between industrial and open source developers by replicating the survey in an industrial context.

4.5.1 Creation of Configuration Options

C1.1 Options are not Planned During the interviews, we learnt that in most of the cases there are no plans or guidelines on which options to add in the source code and where to add them, leading developers to mostly use their own judgement. Our survey (question 11)

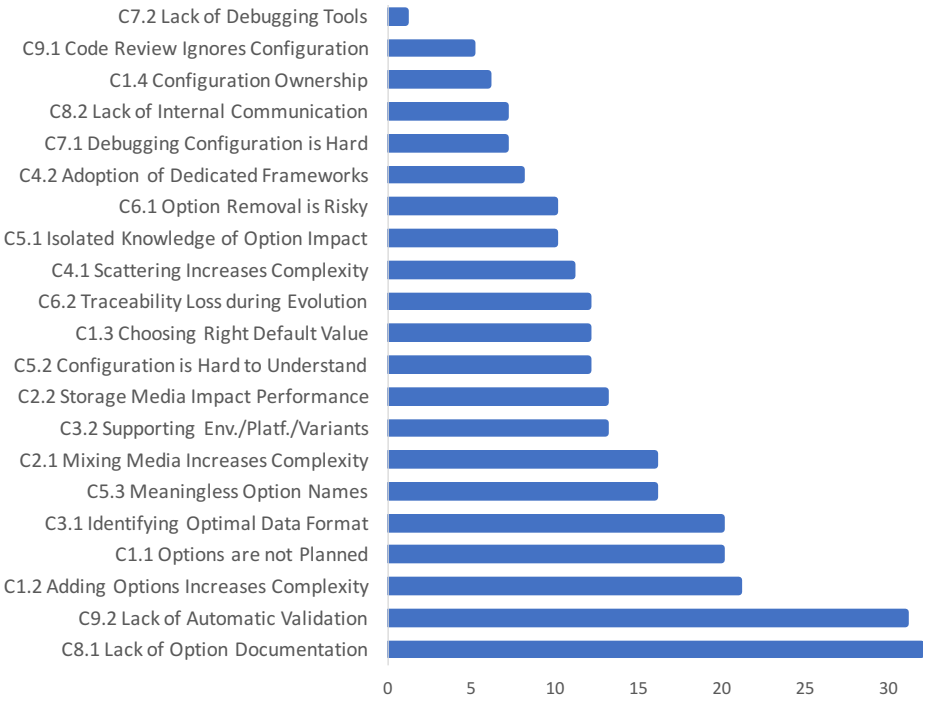


Figure 4.3 The identified configuration-related challenges, ordered by the number of survey responses mentioning them (open-ended survey question 29).

confirmed that 60% of the surveyees add options without any planning, while 40% specify options in the architecture specifications and 24% in the requirements (people could choose multiple answers). In 24% of the cases, new configuration options were meant to externalize a string constant from the source code, since developers typically *"hard code first and externalize later"*. Although our survey is sent to open source contributors, we observed a large number of participants mentioning architecture and specification documents. We think that this is due to the large projects we focused on, in which relying on such documents is a common practice. Furthermore, in practice open source developers, especially in larger projects, often do so in the context of a company who is paying them to represent their company's interests in a project's development.

Although six survey respondents in question 29 mentioned that careless addition of options can "*create [a] maintenance burden in the long term*" and "*They don't presume that others might need them*", the lack of option planning is not necessarily due to wrong developer intentions. For example, 7 respondents explained how it is difficult to decide what functionality should be made optional due to the lack of direct access to users or due to the users not knowing it either: "*The most important problem is knowing what options our users want and how to tailor those options to that audience. Often times we can't communicate with all of our users and that creates a very big gap of understanding between developers and our userbase.*" In the worst case (two survey responses) this can lead to the wrong data format being chosen for options: "*We should have anticipated nested configurations, but the config system sort of grew organically at first. Later we had to step back and redesign it.*"

C1.2 Adding Options Increases Complexity The interviewed experts and 14 survey respondents (in question 29) unanimously confirmed that a large number of options makes software configuration a more complicated task: "*There is no fun to create new options. Options are increasing the complexity*". Apart from complicating the configuration of a software system, it makes the meaning of options harder to understand and hence to maintain: "*I have bad experience with "configuration bloat" which means that sometimes there is a plethora of config option and you don't really need 99% for almost all usecases. In effect this means someone getting acquainted with the software will miss important config options because most of them can be safely ignored.*"

The risk of increasing complexity is even larger in practice as not only the number of configuration options in a file is growing, but "*[the number of] configuration files grows organically*" as well (5 respondents to question 29). In other words, the options are being spread out across more files, largely because developers (especially novices) "*do not immediately see the*

complexity they are adding by creating new options" or configuration files, and because adding new options is largely an *ad hoc* decision that is not planned ahead of time (see above).

C1.3 Choosing the "Right" Default Value Even though options are added to enable end users to customize an application to their needs, 12 respondents stressed (in their responses to question 29) that it is essential to pick the right default values for each option. For 6 respondents the term "right" corresponds to enabling out-of-the-box functionality of the application for the majority of the end users: "*Many users never configure the program, and so choosing the best possible defaults for configuration options is important.*" For 4 others, the default values are even essential to avoid breaking the system: "*Graylog may destroy all your logs if the log retention module's configuration can not be loaded properly, because the default in the code is to have a very short retention period*". Unfortunately, all 99 people responding to question 12, and 12 respondents to question 29 concluded that finding the right default value is a largely *ad hoc* process.

C1.4 Configuration Ownership Figure 4.4 (question 9) shows how 47% of the survey respondents allow any developer to add new configuration options, while in 53% of the cases this is restricted to a group of experts, i.e., a group of developers (24%), architects (5%) or a mixture of developers and architects (24%). Especially in sensitive contexts like banking applications, architects are the main responsible party for configuration creation.

Allowing anyone to add options has several side effects. First, a configuration file can be filled with redundant configuration options configuring the same thing. Second, conflicts between a new option and existing ones generally do not pop up in a developer's individual workspace, but only during integration of the developer's changes with other developers' work: "*[they] had to merge all [modified] configuration files, and hence they got for some cases, more than one option that configure the same thing. This was complicated to fix and consider*". This is mostly because developers in general don't *"have a broad vision on different components and a long-lived experience on the system"* and because a configuration file is an external artefact that could be *"shared between different components and/or different applications"*.

4.5.2 Managing Storage Medium

C2.1 Mixing Storage Media Increases Complexity Having many different storage media in one software system increases the complexity of configuration options, and hence makes them harder to find, use, understand and maintain. One interviewed expert even considers software configuration as a *"neural network"*, because a potentially large number of options and configuration files is distributed across a variety of different storage formats, yet for the outside (i.e., end users and developers), they should look as one coherent set of

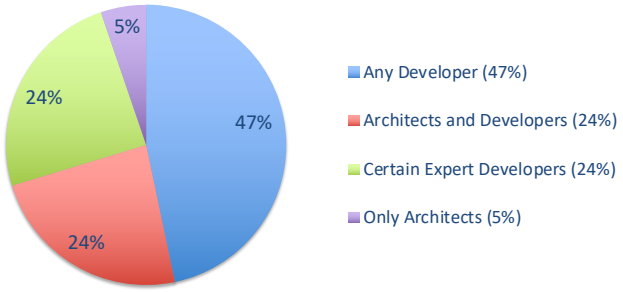


Figure 4.4 Who is Responsible for Option Creation (survey question 9)?

options. Apart from confusion, other negative consequences include difficulties integrating the options from different storage media as well as updating the options in the right location.

Unfortunately, mixing storage media is quite common-place if different options should have different binding time. Six respondents (question 29) also explained how a physical distinction needs to be made between confidential configuration data, such as passwords and secrets, and public configuration data. Hence, at least two storage media are required to achieve such a distinction. Similarly, in larger distributed systems different nodes might require their own configuration storage.

C2.2 Storage Media Impact Performance Surprisingly, there are also performance implications related to the choice of storage medium (and binding time). Seven respondents (question 29) mentioned how the choice of a storage medium with load-time binding time requires reloading the whole application just to refresh one option's value. Five respondents experienced heavy load when using a configuration database with execution-time binding, as their system accessed the database too frequently to check for updated option values, to the point that any down-time of that database would cripple the system's behaviour. Finally, one respondent experienced race conditions, where an application would accidentally read an option's value right before changes would be made, hence missing the updated value.

4.5.3 Managing Option Data Format

C3.1 Identifying Optimal Data Format Identifying the right data format is a major configuration-related challenge, with 10 respondents (question 29) experiencing slowdown due to too weak option type systems, wrong types chosen for an option (e.g., string option not supporting special characters), inability to group related options or lack of support for adding comments to options and their possible values. Three respondents were unable to easily express hierarchical configuration data (i.e., configuration options that are composed of other sub-options), while four were unable to deal with graph or struct option values.

Finally, two people pointed out (question 29) the lack of reuse mechanism in existing data formats. Even though several configuration files could share common lines ("*Consider a configuration file that supplies values suitable for X region and Y environment, and another configuration file that supplies values suitable for Y environment and Z data center*"), one needs to duplicate those lines across all files. An *ad hoc* "include" mechanism seems a possibility, but is not straightforward to enable and maintain.

C3.2 Supporting Multiple Environment/Platforms/Variants As suggested by one of the respondents in the previous challenge, an application often needs to be deployed in multiple environments (development, test and production; 6 respondents), on multiple operating systems or on mobile platforms (3 respondents). Furthermore, different variants of a given application might need to be built, such as a cheap, light version vs. a more expensive, full-featured one (2 respondents). Having to provide a whole folder of configuration files, one per environment, platform and variant is to be avoided due to combinatorial explosion and because "*If you have different config files for each, you can hit problems when you release due to the production config being untested*". Finding a sufficiently expressive data format (and storage medium) avoiding such explosion, or at least making it more manageable, is a common challenge experienced by the surveyed participants.

4.5.4 Configuration Access in Source Code

C4.1 Scattered Access Increases Complexity 7 survey respondents claim (in question 29) that "*from the application perspective the location of the configuration values should not be visible or make a difference*", urging to build code abstractions to hide configuration storage media. If not, understanding configuration use, its maintenance and debugging become more difficult, while any changes in data format or storage mechanism risk propagating across the code base.

Despite 60% of the surveyed developers accessing configuration options from within a single class or module, 40% still read configuration options from multiple storage media in different places of the software system's code (question 8).

C4.2 Adoption of Dedicated Frameworks Similar to Sayagh et al. [163], the interviews and survey showed how developers typically do not tend to use existing configuration frameworks, or only opt for the most basic ones (e.g., the preferences API in the standard Java SDK). Yet, a large number of sophisticated frameworks exists to make accessing options cleaner, scale to large configuration spaces, improve type-safety or reduce memory footprint. It was suggested by one respondent that this is because *"developers are lazy to learn and try new configuration frameworks, they like to use the easiest method"*.

In any case, 66% of the surveyed developers created their own classes and functions to read and manipulate configuration options from configuration files, while only half (49%) of the surveyed developers had used an existing configuration framework in some project. The latter percentage is basically the same (48%) as that of developers using just a basic I/O library (instead of a dedicated configuration library) for reading configuration files (question 6).

4.5.5 Comprehension of Options

C5.1 Isolated Knowledge of Impact of Options Despite some of the interviewed projects using a dedicated configuration expert, no single expert was said to know the goal and impact of all the configuration options of her software system. *"Only the developer knows the meaning of options he created. Configuration file is like a black box for deployers (which are the configuration users)"*. The survey confirmed this (question 19): only 31% of developers know the impact of all configuration options, whereas 46% know the impact of the majority of configuration options, and 23% have only a limited knowledge about the impact of configuration options. The latter people know the impact of only a few options (12%), only the options used in the source code they worked on (8%), or do not know at all the impact of any configuration option (3%). As mentioned by a surveyed participant in question 29, isolated knowledge is a risk: "*The one [who] created that option has quit the team and that option [is] invoked in too many places of the code and hard to guess what it does*".

We also studied the relation between knowledge of options and the number of configuration options. Of the surveyed participants who had indicated that they understand all configuration options in their system, only 11% worked on a system with in between 100 and 1,000 options. The majority (41%) worked on a system with less than 10 options. Hence, the complexity of having more options (C1.2) plays a major role in terms of configuration knowledge. Future work needs to investigate on this direction to find the impact of different factors such as projects size and engineering roles on this challenge.

C5.2 Configuration is Hard to Understand We found that surveyed participants use different techniques to understand configuration options (Figure 4.5). Based on survey question 20, we found that 74% and 52% of developers use documentation or configuration file comments, respectively, to understand the role of a configuration option. Since documentation and comments are not always up-to-date or clear for everybody, 59% and 2% of developers have to use the source code or debugging tools, respectively, to understand configuration options. Code search often is complicated because the option names do not appear as string literals or the option names are dynamically constructed via string concatenation. Instead, we found that 40% of developers ask a colleague for clarification about the goal of a configura-

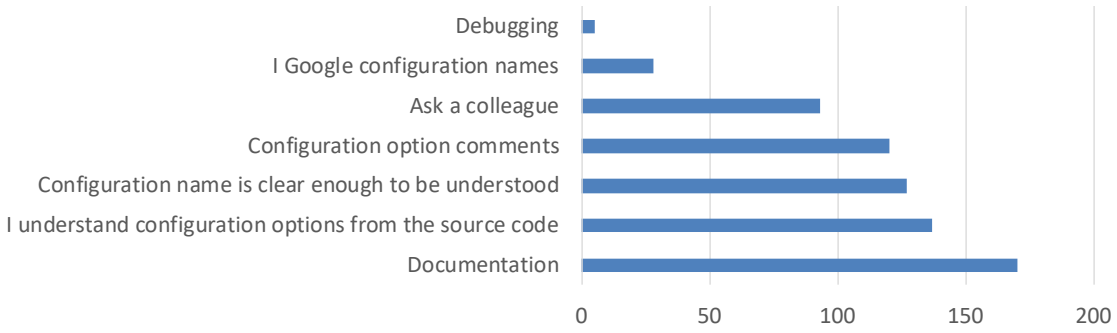


Figure 4.5 Approaches to Understand Configuration Options (survey question 20).

tion option and 12% of developers search online. Only 55% of developers rely on configuration names to understand the objective of a configuration option. Note that this survey question allowed respondents to choose multiple answers.

Despite all these sources, comprehension challenges remain an issue when trying to understand the options of third party libraries (3 respondents for question 29), the changes to options beween different versions of a library, or the interplay between different configuration options. For example, "*Having a stack of configurations, each one overriding parts of the previous ones. One needs to be very clear on the order of evaluation up front and remember to log the resolving.*" Options can depend on each other, override each other or interfere in other ways, all of which render configuration more complicated, especially as the number of options keeps on growing (C1.2).

C5.3 Meaningless Option Names 16 survey respondents highlighted (in their responses to question 29) the difficulty of finding meaningful names for options, where meaningful implies being short, descriptive and easy-to-search in the code base. In many projects, "*Many config options were added early on without any consistency*", but even to "*come up with an overall workable naming convention for all configuration options*" is not straightforward. Based on question 14, we found that only 54% of developers follow a naming convention for configuration options. Of the other developers, 23% have a naming convention but do not respect it, while 22% do not have a configuration naming convention at all.

4.5.6 Maintenance of Options

C6.1 Option Removal is Risky As shown in Figure 4.6 (and based on question 21), only 1% of the surveyed developers frequently change configuration options during maintainance, while 29% of the developers sporadically do so. On the other hand, 62% almost never make such changes and 8% never. As one interviewee put it (with 7 survey respondents making

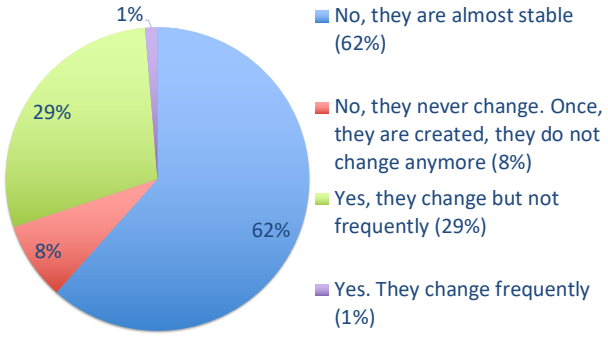


Figure 4.6 How often configuration options are maintained by engineers (survey question 21).

similar claims): "*Cleaning configuration options! Oh no way. No one can take such a risk, we don't clean the configuration files, because we don't know when and where the system can crash. Dead options are kept in the configuration files forever.*"

Furthermore, for two interviewees, developers were not allowed to touch a configuration option due to the fact that these options were included in the official requirements specification, which corresponds to a contract with the client. As such, dead configuration options had to be left in the configuration file and/or in the user configuration UI. For most of the other interviewees, removal of dead options would only happen for options visible to the end user, while developer-only options typically are not a priority for removal.

C6.2 Traceability Loss during Option Evolution Based on question 22, we found that 58% of the survey respondents do not store removed configuration options anywhere, they just remove them from the configuration files. Only 4% use a version control system to track removed configuration options, while 3% store removed options in a database and 12% just comment out unused options in the configuration file.

A related challenge is the need for keeping new options backwards compatible with older ones (4 respondents to question 29): "*how to deal with introducing or removing configuration options in new software releases, considering that the release may have to be rolled back in case of a faulty release. If users have already started using new configuration options, the old release will not recognize them and error out*".

4.5.7 Resolving Configuration Failures

C7.1 Debugging Configuration Failures is Hard Via the interviews, we found that debugging configuration failures can be difficult, especially in multi-component systems, or software systems that depend on other software components or services. For example, one of

the interviewees lost considerable time trying to change an option hosted in another system, managed by another company. Unclear information about the values of configuration options further complicates debugging: "*error messages are horrible, they don't reflect anything about configurations*". 7 respondents (question 29) complained about execution logs not providing sufficiently detailed information to even determine if a failure is configuration-related or not: "Configuration options for things like timeouts retries or other connection options are quite useless without good monitoring".

C7.2 Lack of Debugging Tools The interviewed experts are debugging configuration failures with the same tools as they are using for debugging ordinary bugs. As shown in Figure 4.7 (and based on question 17), the most popular artefacts used to debug a misconfiguration are the failure message (76%), log files (74%), stack trace and the source code (62%). Only 5% of respondents use an automated tool to resolve misconfigured options. Generally, interviewees and survey respondents were not familiar with the many research tools for debugging configuration failures proposed in the literature [44, 46, 47, 68, 69, 104, 120, 136, 147, 164, 169, 178, 192, 193, 193, 195, 195, 204, 216–219]. One respondent summarized the current state-of-the-practice as "*Dig deep and hack till it works*".

C7.3 No Strategies to Avoid Regressions As highlighted by Figure 4.8 (and based on question 18), the most popular artefacts in which developers record information about configuration failures and the workarounds used to resolve them are the commit message (44%), bug reports (33%), and wiki (24%). In addition, we found that 22% do not document configuration failures nor their solutions at all.

On the other hand, in the case of projects who do document configuration errors and their resolution, surprisingly few people use that information. Most interviewees reported about developers losing precious time resolving the same configuration failures over and over again, without realizing that documentation was available about them. In other words, no clear process was in place supporting developers when resolving configuration failures.

4.5.8 Configuration Knowledge Sharing

C8.1 Lack of Option Documentation The interviewed experts mentioned that "*documentation and comments [(in the configuration file)] are rarely made for configuration options*", largely because of limited (time) budget and because developers do not like to spend additional effort on this. To confirm this, we asked survey respondents (questions 23, 24 and 25) to rate the quality of their option documentation and of the comments in their configuration files. As shown in Figure 4.9 and 4.10, 39% of respondents gave a rating of 3/5 for the doc-

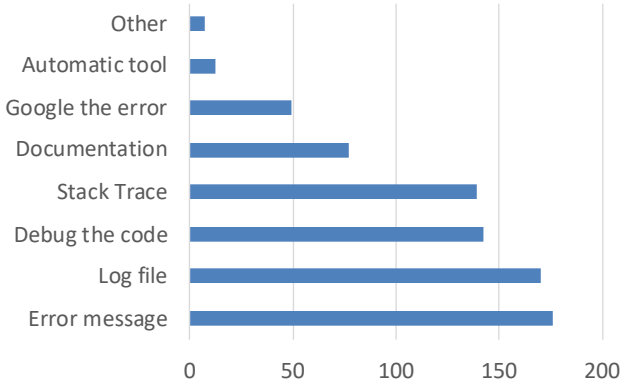


Figure 4.7 Artefacts used to debug configuration failures (survey question 17).

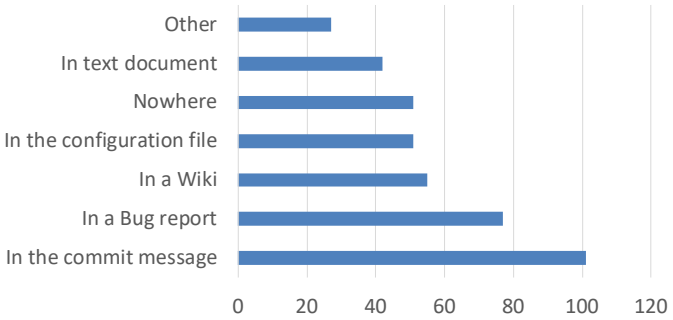


Figure 4.8 How developers document configuration failures and their resolutions (survey question 18).

umentation of configuration options, compared to 31% giving a rating of 3/5 for the quality of comments in their configuration file.

Hence, although developers know the impact of good documentation and comments on configuration, this documentation often either is missing or is incomplete. For example, the presence of good documentation often depends on "*the writer context. The writing style of a documenter can influence the quality of the documentation*". 8 of the respondents provided examples of missing information, such as the goal and impact of an option, its links to user requirements, an explanation of its potential values, pointers to training/FAQ documentation or even an explicit domain model of the options.

C8.2 Lack of Internal Communication Finally, we also found that internal configuration-related communication amongst a development team is rather limited. As shown in Figure 4.11 (and based on question 13), 55% of respondents communicate newly created options via textual documentation, whereas 36% of respondents use the pull request as a way to

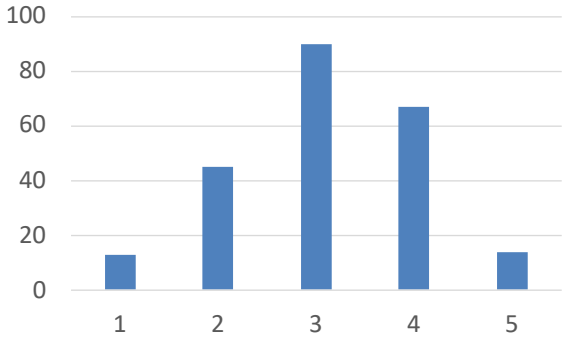


Figure 4.9 Quality of documentation, rated from 1 (low) to 5 (high), based on survey question 23.

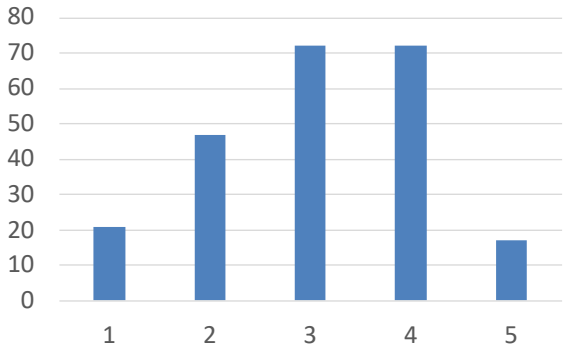


Figure 4.10 Quality of configuration file comments, rated from 1 (low) to 5 (high), based on survey question 24.

communicate configuration options and 32% communicate new options via a wiki or a web platform. On the other hand, 25% of the respondents communicate new options only orally.

Such limited communication has as side effect that developers are not aware of existing options that could apply to their situation (5 respondents to question 29), leading them to add duplicate options: "*sometimes new people don't know that a feature already exists and can be deployed using just a configuration*". Furthermore (2 respondents to the same question), "*In addition to lack of discipline, it is often lack of knowledge in the first place. Chapters on Configuration management from books like Continuous Delivery would help and should be mandatory reading, I believe.*" Such books indeed provide techniques and practices to deal with secure vs. non-secure configuration data (either environment- or application-related), or to communicate between developers and operations (devops).

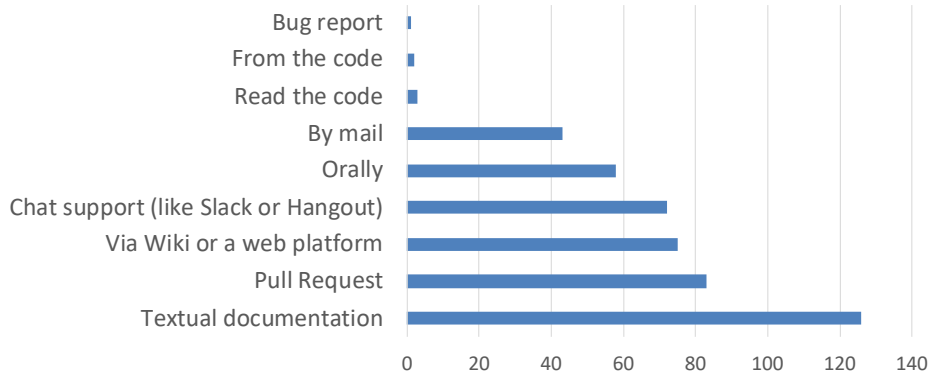


Figure 4.11 Mechanisms used to communicate new options (based on survey question 13).

4.5.9 Quality Assurance

C9.1 Code Review Ignores Configuration The interviewed experts review configuration-related changes only in cases in which a configuration option will be manipulated by end-users, and ignore configuration that are changed by technical engineers like deployers or administrators. According to most of the interviewees, reviewers did not consider configuration options at all, as they either consider options as an external artefact or do not have enough knowledge about them: "*And we have all the routes in a configuration file, the problem was the configuration file was not being reviewed, so some developers were testing and changing the routes and pushing the changes without knowing.*"

Based on question 26, 36% of the survey respondents do not review all the configuration options: 15% review only their application's source code and completely ignore configuration options in their review, 8% do not follow any review process, they do not review neither their software source code nor the configuration options, 5% review only functional options, and 8% review only technical options. We found that 64% review both technical options and functional configuration options. This relatively large percentage seems largely due to the pull request mechanism provided by GitHub (since all survey respondents were identified via GitHub).

C9.2 Lack of Automated Validation Apart from code review, we asked the survey respondents (via question 27) which configuration quality assurance techniques in general they are using. As shown in Figure 4.12, 82% of developers mainly use tests, 19% have tools to validate the correctness of configuration values, while 17% use containers (like Docker) to configure the environment and application once then share the resulting container with other roles, avoiding costly (and error-prone) reconfiguration by everyone. In addition, 14% of respondents use a database to persist the entire history of configuration values and to

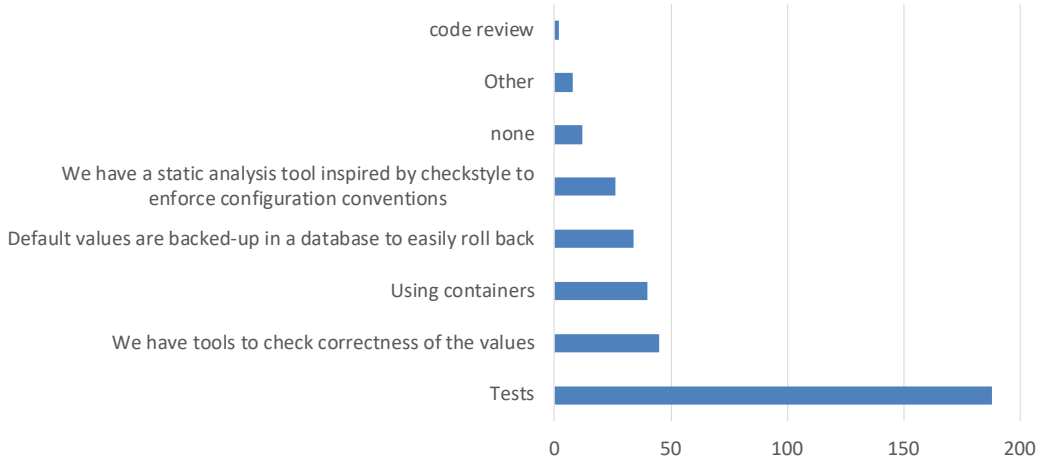


Figure 4.12 Quality assurance techniques used by respondents (survey question 27).

have potential backups, while 5% do not follow any practice to assure the quality of their configuration options.

Offline, load-time and run-time validation tools generally are missing. For example, 10 respondents (question 29) mentioned the need for offline validation tools able to spot simple syntax or punctuation issues, and that could be integrated in the pre-commit hook of their Git repository. 2 respondents (same question) needed load-time validation of the semantics of the option values, while 6 mentioned similar run-time validation tools. The latter should be paired with adequate run-time handling of configuration errors, such as "*extensive bounds checking for configuration options, and storing a default and known-working configuration file inside the application, so that if values are missing, they can be replaced with the defaults. Some error output, letting the user know that they messed something up in the configuration file is helpful as well. Sometimes I've gone so far as to point out exactly where (down to the very character) a configuration error was found.*"

Finally, 12 respondents (question 29) mentioned the lack of testing tools of options, in particular tools allowing to test a code change across all relevant configurations instead of just for one option (value) at a time: "*The existence of options implies an additional testing burden, and even when it is possible in principle to test a given option (it does not require any special operating system, for example), often it is not tested, or not all combinations of interacting options are tested.*"

Table 4.3 Mapping the challenges (C1.1 to C9.2) to the recommendations (R1.1 to R9.5). The symbols + and - respectively indicate positive and negative impact of a recommendation on a given challenge (i.e, recommendation R1.1 positively addresses the challenges C1.1, C1.2, C5.2, and C6.1).

	R1.1	R1.2	R1.3	R1.4	R2.1	R2.2	R3.1	R3.2	R4.1	R4.2	R5.1	R5.2	R6.1	R6.2	R7.1	R7.2	R7.3	R8.1	R8.2	R9.1	R9.2	R9.3	R9.4	R9.5
C1.1	+										+							+						
C1.2	+	+	+										+									+		+
C1.3				+																+	+	+		
C1.4		+																+						
C2.1					+	+		-+		+												+		
C2.2					+																	+		
C3.1							+	+		+												+		
C3.2					+					+	+									+	+		+	+
C4.1							+		+			+	+									+		
C4.2										+								+						
C5.1		-							+		+	+	+	+				+	+					+
C5.2	+		+		+	+	+			-+	+	+	+	+	+			+	+			+		+
C5.3		+	+	+	+						+			+				+				+		
C6.1	+	+	+		+				+	+	+	+	+	+						+	+	+		
C6.2					+				+		+			+		+		+						
C7.1			+		+		+		+		+	+	+		+	+	+			+	+			
C7.2															+		+			+	+			
C7.3														+										+
C8.1				+							+		+			+		+	+					
C8.2		+												+		+		+						
C9.1		+							+		+									+		+		
C9.2									+	+	+				+					+			+	

4.6 Expert Recommendations

While the previous section discusses configuration challenges faced by practitioners, this section provides recommendations by both open source and academic experts to address these challenges. They are obtained firstly from the interviewed and surveyed practitioners. In the two open questions 29 and 30, surveyed developers were invited to talk about a major configuration-related problem they faced in their experience and how they fixed it. They were also invited to recommend three good practices to follow and three bad practices to avoid when working with run-time software configuration in general. We enriched these practitioners' recommendations by academic approaches and findings identified during our systematic literature review (see Section 4.3).

A first goal of this section is to inform practitioners not only about how their colleagues have dealt with certain configuration challenges, but also about the state-of-the-art in academic literature. The second goal of this section (and the mapping in Table 4.3) is to emphasize areas that are not covered enough by literature, opening up avenues for validation of practitioners' recommendations along with new topics of research on promising technologies and methodologies to resolve important configuration challenges.

4.6.1 Creation of Configuration Options

R1.1 Treating Options as Scarce Resource In order to avoid having "*configuration files filled with useless information*", 66 survey respondents recommend to limit "*the number of options to an absolute minimum*". This is also discussed in the literature by Xu et al. [212], who found that end users change just a small portion of a software system's available configuration options, and hence developers should improve the design of their software system's configuration by presenting end users a minimal set of options to configure.

Developers should try to avoid adding new options, especially if they can "*determine the proper value by calculation*". Options should only be added if "*a major use case is unrealizable without one*". A second surveyed developer confirmed that one should use a planned and defined strategy to create new options: "*every configuration option has to be supported by a use case. Even [then,] if you do not have [an explicit] planning, if you can not think of a valid use case that the configuration option will [fix] that can not be [fixed better] by another solution, do not add it*". One should "*plan in advance the structure of the configuration, discuss it with colleagues, listing pro and [cons] of each approach and compare; ponder on what is gained with the new configuration; agree on what configurations will be needed in architectural/design phase*".

Reducing the number of configuration options is also important from the end users' point of view: "*Think about who is going to use the software, how they will want to use it, and how many ways it will be deployed. Does it need to just be for advanced users that know what they are doing? If so, it should be flexible with lots of configuration with sensible defaults with excellent documentation of the options. Will it be a consumer product that should work in most cases? In that case, think much more carefully about what you make configurable because you'll have to support misconfigured deployments and you don't want to confuse customers*".

R1.2 Assigning Option Ownership Apart from reducing the complexity of software configuration by limiting the number of configuration options and files, two of the interviewees stressed that they also limited the number of developers allowed to touch configuration options and files, even to only one developer in some cases. Any other developer willing to add, modify, or remove a configuration option should get the responsible developer's approbation. This could, for example, improve the names chosen for options, positively impacting challenge C5.3, as well as reduce the number of redundant and unnecessary options (challenge C6.1). Limiting the number of people allowed to manage configuration options is also suggested by 5 surveyed developers (question 30).

At the flipside, if one of the few responsibles leaves or would not formally document his or her options (cf. challenge C5.1), the recommended practice of clear ownership backfires. This is why Table 4.3 shows a negative impact of R1.2 on the C5.1 challenge. Therefore, it is important to trust the right developers for this task. One mitigation strategy presented by an interviewed expert is to follow a *"pair programming"* practice, where configuration changes are always made by two developers. Apart from increasing quality assurance, this practice automatically ensures that whenever one of the developers leaves, at least the second developer is still around to take over (and start pair programming with a third developer).

R1.3 Making Configuration Cohesive 8 survey respondents suggest that developers should "*make each configuration option responsible for one single thing, do not reuse the same option for different cases"*. In addition, they should never "*create multiple options that have the same role. [This problem is] not likely to happen in a small/new project but it could be the case in big projects"*. Therefore, a project team needs to *"keep all the configurations in check"*.

As potential side effects, it will be easier to define an explicit name for an option that has only one precise and explicit purpose. This is why we mapped R1.3 as a recommendation for C5.3 in Table 4.3. Furthermore, when an option is made for one explicit goal, its impact on the source code is minimized, which simplifies the removal of an option (challenge C6.1).

R1.4 Selecting Out-of-the-box Default Values 18 surveyed developers agree that one should choose "*proper default values*", which makes a software system work correctly from the first run: developers should never "*expect users to modify configuration for a first run*". This is mainly because "*a lot of [...] users will not change options and all of them will expect your software to work well with the defaults*". Furthermore, it might require quite some effort for them to find the optimal configuration for their deployment.

While these practitioner suggestions are not as concrete, researchers have explored this topic in far more detail. A large body of work tries to model the configuration of an application, then use this model to suggest an optimal configuration, for example using non-linear regression [81], Markov decision processes [58] or Plackett & Burman's statistical approach [63]. Other work [109, 171, 207] formulates the selection of configuration values as an optimization problem for which iterative search [109], multi-objective optimization [171] or smart hill climbing [207] can be used. While the above models all are offline models, Dia et al. [66] proposed an agent that automatically adjusts configuration option values of a generic application at run-time to guarantee CPU and memory usage objectives.

To improve the scalability of the above models and optimization algorithms, a lot of work has explored how to reduce the search space of configuration values. While a boolean con-

figuration option by itself only has 2 possible values, a combination of 5 such options leads to 32 different combinations, which only gets worse when options have an integer or even string type. Thonangi et al. [183] proposed an adaptive search algorithm to minimize the space of configurations to explore in order to find an optimal configuration, while Siegmund et al. [170] use a threshold-based approach, which consists of detecting interacting options that significantly improve an application's performance.

A more common technique for reducing the search space are sampling algorithms, which select a reasonable subset of configurations, either to build a model or as a more targeted starting point for optimization. Osogami et al. have proposed different sampling heuristics [134, 135], while Sarkar et al. [156] used progressive and projective sampling. Progressive samping starts with a small sample set of configurations and each of their associated measurements for building a prediction model, then progressively increases its size until reaching an optimal sample size and prediction performance. Projective sampling starts by generating a set of initial points in the prediction learning curve from which it projects the optimal sample size to build an accurate optimal configuration prediction model. Two heuristics were proposed by Sarkar et al. [156] to improve the projective sampling, i.e., t-way feature coverage (changing t option values at a time for each test) and feature frequencies (each feature should be selected at least once in the initial sample). Duan et al. [70] used an Adaptive Sampling algorithm that selects experiments to run in order to find an optimal configuration. Zheng et al. [223] proposed MassConf, which collects a software user's configuration and environment information, then uses it to propose an optimal configuration for that user.

Only one research paper provides explicit guidance on selecting configuration values. However, its guidelines are specific to the configuration of garbage collectors in programming languages. Gousios et al. [79] found that such garbage collector configurations have a substantial impact on the performance of server applications. One of their recommendations is to calculate an application's memory allocation rate and its object sizes to adequately configure the Java garbage collector.

4.6.2 Managing Storage Medium

R2.1 Minimizing the Number of Storage Media In addition to reducing the number of configuration options, 31 surveyed developers believe that it is also important to minimize the number of configuration files and mechanisms. Developers should indeed "*consolidate configuration options in as few files as possible*" and should avoid having "*too many floating configuration files*". Several surveyed participants recommend to create "*a central file with all default options*".

Minimization is especially important when considering that projects typically have different variants of their configuration file, one per environment (e.g., production, test or staging). The less configuration files, the less variants of these files there will be, hence minimization favourably impacts challenge C3.2 in Table 4.3. Similarly, a minimal number of storage media makes traceability of those media and their values easier (challenge C6.2).

On the other hand, one interviewed expert mentioned that there is no best choice for configuration storage medium. He suggested that a team should carefully discuss which medium should be adopted in advance. This is not only by considering the developers' preferences, but also by considering the existing technologies for reading and manipulating options.

R2.2 Organizing Configuration Options 20 surveyed developers suggested to clearly organize configuration options according to a uniform abstraction. One should "*avoid mixing 3rd party configuration values with application specific configuration*" in the same configuration file. However, organizations should not overdo it. They should avoid using "*a domain specific language for configuration [...], don't make your user learn another language just to configure your application*", just "*use a standard [file] format*". This is indeed a popular topic discussed by practitioners on Stack Overflow [13].

To abstract away from the specific storage medium (and data format) used, several software systems provide configurators, which are tools that help users easily configure their software system. However, such configurators typically are built *ad hoc* and suffer from a low quality. Therefore, Perrouin et al. [139], Abbasi et al. [36], and Boucher et al. [52] proposed approaches to reverse-engineer these configurators into a model of the configuration's data format (i.e., the configuration options with their types and constraints), then generate more consistent configurators from that model. Behrang et al. [50] presented an approach to find inconsistencies between user interface configurators and source code via static analysis. Evaluation of their approach on Mozilla Core and Firefox was able to find 40 real inconsistencies.

4.6.3 Managing Option Data Format

R3.1 Using Simple Option Types 9 developers recommend to simplify the types of configuration options as much as possible in order to improve the understandability of these options (challenge C4.1). Boolean configuration options are indeed much easier to configure compared to string configuration values. Based on empirical analysis, Xu et al. [212] recommend developers to simplify their configuration option types by explicitly identifying the configuration values used the most by their users. Similarly, developers should improve the error messages generated upon "*typing errors*", since it is "*not always clear if values are*

integers/floats/strings and can result in problems". In addition to defining simple data types, developers should "*be prepared for weird configuration values from users*".

Configuration option types are not limited to generic types like boolean and integer, but typically include more specific types like IP addresses or file system paths. Identifying such types is important to easily configure a software system and prevent errors, but this is not straightforward. One approach to identify such types is proposed by Xu et al. [214] based on a classification tree and a keyword-based method. The latter method exploits the naming convention used by configuration mechanisms to deduce the semantics of an option. Xu et al.'s keyword dictionary has been built by manually analyzing 1,000 options of real open source projects.

R3.2 Identifying Sensitive Configuration Data While one should select good default values to enable key-in-hand software deployment, one should not ignore the security aspects of a configuration. As suggested by 4 surveyed developers, sensitive configuration data should be secured: one should "*not store passwords and other sensitive parts of configuration in VCS*" (Version Control System). In addition, developers should find a way to "*mask sensitive data and securing it in production*". While using separate storage media enables more secure, encrypted storage of sensitive configuration options, it does increase the number of media to look at in order to understand a system's configuration. This is why we mapped R3.2 to C2.1 in Table 4.3 as "-+".

To enforce the security aspect of sensitive configuration data, Sun et al. [179] proposed to automatically generate access control configuration based on access control requirements. In other words, this approach goes from a specification that indicates which role could perform which operation and under which conditions, and then generates access control configuration of J2EE applications. In addition, Wang et al. [194] proposed an approach to identify and reverse engineer access control configuration options, which allow or deny users from accessing a resource, e.g., a file, folder, or URL. Their approach consists of analyzing configuration files to extract configuration options, and uses taint analysis on the source code to understand how it interprets configuration options, and in which order it verifies the value of each these options. The goal of these analyses then is to generate a set of rules that define the relation between users, permissions, and resources. These rules can then be used to detect security policies defined in a configuration file.

4.6.4 Configuration Access in Source Code

Once developers decide to create a new configuration option and add it to a storage medium (e.g., a configuration file), they need to use this new option in the source code. Below, we provide a set of recommendations about how to use options in the source code.

R4.1 Encapsulation of Configuration Access in Single API 27 surveyed developers discussed how to use and access configuration options in the source code. The major recommendation from these discussions is to "*prevent reading of options from too many places in the code*" and hence to avoid accessing directly the configuration options from all over the code base. In addition, surveyed developers suggest to "*use a single strategy [a]cross the application*" to read options, and hence not to mix-and-match different APIs for doing so.

In particular, surveyed developers suggest to "*try to centralize the configuration in an API*", which should not be "*too verbose (e.g. Config.getInstance(getValue(Config.KeyType.X, ConfigKey.ValueType.STRING, default_value) because it's a pain to read*". Developers should "*make sure the configuration API is well understood*". Apart from making it easier for developers to access configuration values, a specific API also limits the number of methods in which load- and run-time validation of configuration options can be put (challenge C9.2 in Table 4.3).

R4.2 Adopt Existing Configuration Frameworks 13 developers suggest to use an existing configuration framework or library instead of reinventing the wheel by making a new framework or API. A good framework can help developers abstract away from low-level configuration option access, in the sense that the framework can automatically identify where an option is defined and stored (challenge C2.1). Furthermore, a framework that is easy to use can help developers in navigating the source code and hence understand the impact of an option in the source code (challenge C5.2). For this reason, some developers specifically suggest to use a framework leveraging annotations and configuration value injection, which reads configuration option values then injects them in the right source code variables and attributes. Various developers explicitly stressed that developers should take the time to "*familiarize themselves with the mechanisms made available by the framework so that they are able to select the most appropriate mechanism*".

While configuration frameworks are recommended by practitioners, the choice of a specific framework is not straightforward, since there are dozens of (open source) frameworks on the market. Denisov et al. [65] compared three major Java configuration frameworks, while Sayagh et al. [163] analyzed the popularity of 11 Java configuration frameworks and the important factors to consider in choosing a suitable configuration framework. These studies

identify a (potentially large) set of configuration framework characteristics to consider in choosing a suitable configuration framework, including how actively a configuration framework is maintained by its developers and how well it is documented.

4.6.5 Comprehension of Options

R5.1 Explicit Option Naming Convention 67 developers think that having a good configuration name is essential to improve the quality of a software configuration. An option name should be sufficiently "*clear and descriptive*" to be understood by end users. In addition, configuration options should be "*structured [...] from the start, probably split them by responsibility/modules/plugins/etc - whatever fit[s] your project best*".

Explicit naming convention could also include an indication about the environment used for that option (challenge C3.2). For example, one can distinguish between the environment of two options from their names like "*development.database.username*" and "*production.database.username*", where the first option is for the development environment and the second one concerns the production environment. Similarly, the configuration name can provide an indication of the plugin or component it covers, making it easier for developers to identify where an option can be used and for what feature (challenge C5.1), but also to remove an option entirely from the code base once it is no longer deemed useful (challenge C6.1).

R5.2 Comprehension from Code For challenge C5.2 in Section 4.5.5, we found that 52% of developers try to understand the meaning of configuration options by perusing the source code. Without automation, such code analysis is considered to be a tedious job. For this reason, several techniques are proposed in the literature to automatically map configuration options to the source code that they impact. Lillack et al. [114] used taint analysis to track under which Android configuration option(s) a code fragment could be executed. The authors found that their approach works especially well for options related to network and storage, since those options rarely interact with each other. Identifying which options control a source code area can help developers identify options that are rarely accessed in the source code, in order to perform cleaning and refactoring (C6.1). In addition, one could use the approach of Siegmund et al. [169] or Zhang et al. [222] to identify which options have what impact on software performance, which can help on debugging performance configuration errors (C7.1).

4.6.6 Maintenance of Options

R6.1 Pro-active Dead Option Detection 26 surveyed developers believe that one should "*clean out obsolete configuration immediately otherwise it hangs around after everyone has forgotten what it was for in the first place*". In effect, dead configuration options are considered as a form of technical debt in a code base, similar to the observations made by Rahman et al. [150] about feature toggle maintenance in Google Chrome. Those are options that are used during development to enable new features on demand for testing and release, and that should be removed from the code base once the features are stable (making them permanently on).

Note that, apart from dead options, unused options are undesirable as well and should be actively maintained. The latter options are options that are being used (as opposed to dead options), but whose value is not changed often by users. Hence, such options should be "internalized" again, i.e., be turned into a constant value.

R6.2 Limit and Trace Configuration Changes While cleaning and refactoring options is highly recommended, developers should do so with care, as configuration options are sensitive elements of a code base that are hard to test. Developers indeed should not "*change the name of a configuration key unless what it does is changing*", and especially "*between two major versions*". As suggested by one developer, it is important to "*not change configuration without understanding what it does and ever ask an experienced programmer what to do*".

If there is no other choice, and an invasive change needs to be made that risks to break compatibility, developers should explicitly manage the traceability of the configuration options involved. As suggested by 9 surveyed participants, developers should "*Keep track of what [they] did*" and "*keep working versions for rollback*".

4.6.7 Resolving Configuration Failures

R7.1 Right Granularity of Execution Logs 11 surveyed developers suggest to log configuration information and generate meaningful error log messages in case of configuration failures. Developers should "*log when/from where a configuration file is used*". In case of configuration failures, a software system should fail appropriately "*if a configuration is done wrong, or a configuration value is [missing]*". In particular, a software system should not "*silently fail or give an ambiguous error message about the bad configuration. The more details the user can be given, the better*". A suggestion for an error message could be: "*this failed because setting X was bad. Change the setting X in file Y to value Z to make it work again*".

Zhang et al. [221] proposed ConfDiagDetector, which injects configuration faults in a software system to analyze (using natural language processing) the quality of how the resulting failures are being reported or logged. This can be used to help practitioners evaluate the granularity and quality of their logs. Better error logs also help build more powerful configuration debugging tools (challenge C7.2).

R7.2 Document Configuration Failures and Resolutions To avoid regression of configuration failures (i.e., the same configuration failure re-occurring), 4 interviewed experts mentioned that documenting a configuration error and the way it was resolved previously is mandatory. Such documentation does not necessarily need to be formal, since several developers recommended to document configuration failures and their fixes in a project's wiki.

Documenting configuration failures is also important to build up a sufficiently rich data set for building models that can predict whether a bug report is related to configuration. Such models can reduce debugging effort by focusing on configuration options only instead of on the entire source code. Examples of such models are proposed and evaluated by Xia et al. [208] and Bowen et al. [53], which focus on predicting if a bug report is related to configuration. In addition to predicting if a bug report is related to configuration, Wen et al. [200] also predict which option is misconfigured.

R7.3 Automated Configuration Failure Debugging A wealth of research results are available on techniques (highlighted in bold) for various aspects of automated configuration failure debugging. In other words, this is by far the most researched configuration challenge/activity. Despite this, most of these approaches are not known or used in practice according to the interviewed and surveyed engineers. Here, we discuss the papers that are the most closely related to the configuration activities and challenges. We refer elsewhere [185] for a more detailed and technical survey on such debugging approaches.

Whitaker et al. [204] proposed one of the first approaches to debug configuration failures, which aims to **identify the moment on which a software system changes from a working to a non-working state**. Once this transition is identified, the approach checks all modifications made to the initial (working) state to find the culprit option. Similarly, Otsuka et al. [136] compare each configuration value with its past (in)correct values by differencing configuration values before and after a failure. Siegmund et al. [169] build models that describe the impact of a configuration option on software performance. Such models can be used to narrow down the scope of options to debug in case of performance bugs.

Other research efforts use **static or dynamic source code analysis** to identify the root cause of a misconfiguration. Dong et al. [68, 69] use a backward slicing technique that starts

from a failing line in the source code and a forward slicing that starts from the lines reading configuration options. When both slicing techniques meet, the overlapping code paths are analyzed for configuration options. Rabkin et al. [147] use static data flow analysis to map each source code line to the configuration options that may impact it.

Attariyan et al. [46, 47] use **dynamic control and data flow analysis** to identify the root cause of configuration failures. In other work [44], they assign a performance cost to each source code block, then use (dynamic) taint tracking to detect configuration causes of performance failures. Zhang et al. [217, 218] instead profile the misconfigured system, then compare the execution trace with a set of profiles of correct execution runs. Zhang et al. [219] use dynamic profiling to instrument two software versions and execute both, then statically compare the two execution traces to find the culprit option. Attariyan et al. [45] proposed an approach to detect configuration failures by using a healthy execution environment (no configuration failure) as an oracle of correct execution.

Instead of analyzing only the source code, Wang et al. [192] proposed an approach that relies on **user feedback**. The approach takes a failure as input, proposes a fix to that failure from an initial set of sorted options obtained from an existing approach (i.e, those discussed previously), gets feedback from the user to understand whether that fix actually resolved the failure, then adjusts that fix's priority based on the user's feedback. Su et al.'s AutoBash [178] also relies on existing user experience to fix configuration failures. It observes the actions followed by a user to fix a configuration failure, including the actions made to fix a failure and the tests executed to verify that fix's effectiveness. AutoBash then saves a collection of solutions (actions and tests) from different users, and tries them one by one until successful.

While reliable configuration failure data (e.g., logs) are essential to debug configuration failures, **having too much data actually complicates the debugging process**. To easily debug configuration failures in the presence of huge amounts of trace data, i.e., execution traces of system files, registry, and process operations collected from different users of a system, Mickens et al. [120] proposed an approach that relies on decision trees. From a trace of configuration actions (reading or writing files and registry entries on a Windows system), the authors train a decision tree in which each node represents a file or registry key and the edges (or the decisions) decide to read or not a particular option. The leafs of the decision tree represent software exit codes, allowing the user to understand the circumstances leading to a specific failure message.

Debugging of registry configuration failures of Windows applications is another popular research topic. Kiciman et al. [104] proposed an approach to use existing snapshots of correct registry configurations to automatically recover (correct) constraints between reg-

istry configuration keys. Violations of the recovered constraints then indicate the presence of configuration failures. Yuan et al. [216] instead generate a database of rules from the successive events in traces from registry access, which can be used to detect inconsistent events and violations that can identify the cause of a configuration failure. Wang et al. [193, 195] instrument a Windows application to find which registry key it is using, then compares those against a database of existing configuration snapshots. Finally, Strider and PeerPressure follow a trial-and-error approach to identify (and fix) a misconfigured option. While Strider [195] requires manual identification of incorrectly configured ("sick") machines, PeerPressure [193] relies on Bayesian estimation to identify sick machines and snapshots.

While most of the above work considers configuration failures in a specific component or layer of a software deployment, or considers the whole system as a black box, a more recent area of configuration failure debugging research started focusing on **failures that cross the boundaries of software components and stack layers** [95, 162, 164, 197]. Jin et al. [95] found that modern software systems are developed with multiple programming languages and hence need configuration tools that help debug configuration across those languages. Chen et al. [197] found that to resolve cross-component configuration failures, researchers should understand how different components communicate with each other and which configuration options of different components configure the same aspect and hence can cause interferences. For example, both the PHP configuration option *mysql.max_persistent* and the MySQL option *max_connections* configure the amount of allowed connections to a MySQL database, yet it is hard to debug the configuration of both systems to find the culprit of a global configuration failure.

Sayagh et al. [162] analyzed the relation between different layers of the WordPress application stack, and found that an option in one layer could have a substantial impact on other layers. For example, up to 85.16% of WordPress configuration options are used by at least two different WordPress plugins (layer on top of WordPress layer), which can cause serious inconsistencies between different plugins when such "shared" options are changed in incompatible ways. The authors also found that 45% of cross-stack configuration failures are responsible for crashes in production and that debugging such failures requires at least as much effort as single-layer configuration failures. They proposed a modular source code analysis to help debug such failures [164], which basically fuses slicing graphs of individual layers by mapping function bodies to their corresponding calls from higher layers.

To complement the above automated debugging analyses, a number of researchers performed qualitative analysis on configuration failures to **extract recommendations for developers and users to avoid such failures** in the first place. Yin et al. [215] conducted a

comprehensive study on the characteristics of configuration failures including their causes, types, impacts, and how software systems react facing a configuration failure. For example, they found that up to 53.7% of configuration failures are caused by options that violate a predefined format, motivating the development of configuration checkers that verify how well configuration options respect such formats. Similarly, Arshad et al. [43] studied configuration failures in the GlassFish and JBoss Java EE servers, their types, when they occur, and their manifestation (silent vs. non-silent failures). They found that 89% of configuration failures manifest silently and 91% of these failures require a source code modification. They also proposed "*ConfInject*", an approach and tool to inject configuration faults in a software system to evaluate how resilient it is. They used "*ConfInject*" to compare the resilience of GlassFish and JBoss web servers, and found that JBoss performs better than GlassFish. Han et al. [83] found that 59% of 193 analyzed performance bugs are caused by configurations faults. They recommend to identify configuration options that are more likely to cause a performance degradation and prioritize them during performance tests, to test software in a system closely similar to the production environment, and to rely on profiling to debug performance configuration failures.

Finally, once an option has been found to have an incorrect value, a number of approaches have been proposed to **suggest a correct option value**. Swanson et al. [181] try to propose a correct option value relatively close to the current (buggy) configuration using Firefox' configuration constraints (feature model) and a sampling algorithm (e.g., the n-hop algorithm [77], random sampling or covering array sampling). Xiong et al. [210, 211] proposed an algorithm that automatically generates range fixes for a violated constraint. Such fixes correspond to a range of correct values for one or multiple misconfigured options, from which a user can choose correct option values that respect his or her software system's configuration constraints. The approach starts from a misconfigured configuration and the software system's configuration constraints, then generates a range of possible values that each configuration option should have. Evaluation on Linux and eCos showed that the approach could find the range of fixes within a second.

4.6.8 Configuration Knowledge Sharing

R8.1 Communication between Developers and Users 8 developers highlight the importance of communication to define better configuration options. First, developers have to discuss configuration options before starting to code, defining their defaults, ranges and restrictions. Further, one developer proposes to "*let everyone discuss about the configurability needs*", then "*work with the client to understand what they will need to change*".

5 surveyed participants mentioned the importance of sharing configuration modifications. Developers should *"make sure all team members are updated about configuration changes"*, not only developers that should be notified by the new changes, but also the software users. This comprises newly created options, modification of option names and their default values, and also "*how it affects them*". Such openness of knowledge makes it much easier to find the best "owner" of a configuration option (challenge C1.4 in Table 4.3).

R8.2 Pragmatic Usage Documentation The most popular theme in our survey, discussed by 87 developers, is related to documentation. In short, it is "*extremely important [...] that users are not going out of their way to find out what an option does*".

One should "*have a proper documentation above each configuration option in the config file to know what that option does and how does it work in details*". One should document why an option is defined and why it should be added, because "*some users may not be using the software as you are so that the use cases/workflows that make sense to you, may not fit them*". Default values should be detailed in the documentation, with concrete examples. Furthermore, interactions between configuration options should also be specified in the documentation.

Some developers recommend to indicate explicitly the locations where configuration documentation should be added. In addition to the comments added in the configuration file, it is recommended to document options in the project's README file, in the official project documentation, and also in the source code, in order to help developers easily identify where an option is used. One surveyed developer also suggested to automatically generate documentation based on the configuration information in the code base. Only in two interviewed cases, an architect mentioned that the developers have an explicit guideline that forces them to document configuration options. One of these two interviewees' company has an entire team focusing on documentation, including configuration-related documentation.

While we did not find any paper mentioning how configuration options should be documented, Murakami et al. [123] propose an approach that helps manual creators to create documentation. By exploiting the commit logs, which provide the line(s) changed in a configuration file, and a pre-prepared explanation of that modification, the approach generates an HTML manual that can then be edited manually.

A second approach that could be used to help developers understand options and hence create documentation, is the approach of Zhou et al. [224]. It tries to recover the mapping between a configuration option and its associated variable in the source code, which then allows the mining of the option's potential values and constraints. To maintain the mapping between configuration options in the source code and their associated documentation, Dong et

al. [67], Rabkin et al. [148] and Zhou et al. [224] propose static analysis that helps developers automatically identify where each configuration option is used in the source code.

4.6.9 Quality Assurance

Finally, surveyed developers and our systematic literature review propose a set of recommendations to improve the quality of configuration options.

R9.1 Validation of Specific Configuration 25 surveyed developers discussed how to assure the quality of a software configuration by validating the correctness of configuration values. The proposed recommendations differ in terms of the time at which validation is performed.

Many surveyed developers recommended the use of offline configuration validators: "*having an open key/value configuration system is flexible but some of the most difficult configuration errors are of the typo variety (numReceivers vs numRecievers - typo difficult to catch). Would ideally have validators that would warn on unexpected configurations*". This recommendation is aligned with the work of Jha et al. [91] on "ManifestInspector". It identifies errors in an Android manifest configuration file by statically parsing a manifest and validating it against a set of rules stored in a database. Such rules are basically a set of possible XML tags with a set of possible attribute values. They were able to identify 59,547 errors in 11,110 of the 13,483 analyzed Android apps. Eshete et al. [72] build a model that checks the correctness of configuration options in web applications against a "Gold Standard" of configuration values that they built from online discussions, documentation, and expert opinions. Similarly, Zhang et al. [93] detect configuration errors based on a sample of configurations that are enriched with environment configurations.

Several developers recommended validation of configuration values at application boot-time. For example, developers should "*avoid making applications un-bootable in the absence of non-default configuration*". In addition, a surveyed developer suggest to "*do not check logic at run-time to avoid performance penalty, but do check simple value at loading/parsing/importing setting to configuration, accept only valid value*". Configuration value validators provide quick feedback of bad configurations (challenge C1.3) and also support reviewers in identifying incorrect option values (challenge C9.1).

No approaches were suggested or identified for run-time configuration validation. Instead, the most important feature of run-time configuration options that was discussed is the challenge of changing an option value without restarting the software system. A bug could lead to using the old values of these changed options and hence to miss the users' configuration

updates. To fix such problems, Toman et al. [186] proposed an approach based on dynamic taint analysis. It consists of tracking both the usage of a configuration via data propagations and the modification of configuration options in the source code, then checking which version of an option is used in each statement that uses that option. Comparison of the version used by that statement with the last updated version then allows to find inconsistently updated options.

R9.2 Configuration-aware Testing In addition to validation of configuration values, developers also suggest to implement automated testing for configurations. Developers should "*not rely on manual testing of selected few configurations*". Apart from being able to support development of new configuration options, such tests also help to detect configuration regression bugs (challenges C7.1/2).

10 surveyed developers further suggest to adopt explicit testing strategies on configuration options. They basically propose to consider configuration options in integration tests or unit test suites. In addition, one developer proposes "*to have a clear configuration modules for integration testing with the appropriate mock framework, and integrating by using [CI] tools like Jenkins*". In one interviewed expert's context, developers use mutation testing [132] for configuration options.

Configuration-aware testing is the second most popular configuration activity in research (right after configuration failure debugging). Gao et al. [76] proposed a guideline to help developers test their software system in the presence of a configuration subsystem. They represent the configurability of the system as a semantic tree, which can then be used by developers to manually determine the conditions under which to test their software.

Most of the studies, however, focus on **reducing the number of tests to run in highly configurable software systems**, since ideally all possible combinations of all options should be tested. Most of the sampling algorithms discussed for R1.4 could be applied here as well. Bestoun et al [38] use the Cuckoo search algorithm to optimize the number of configurations to test by sampling the most relevant inputs and configurations. Qu et al. [142, 144, 209] propose an approach based on "combinatorial interaction testing techniques". Such techniques sample a set of configurations to test from all possible combinations of configuration values, yet are not feasible for configuration-aware testing due to the large number of possible configurations. Hence, Qu et al. select a specific subset of configuration options to test between two versions. Huning et al. [89] propose a fuzzing technique to test a software system against vulnerabilities that occur only under certain configurations. Marijan et al. [115] built TITAN, which optimizes a test suite for highly configurable software systems using combinatorial interaction, a constraint-based approach to minimize tests, and test prioritization for

regression tests. Fouche et al. [74] used the coverage arrays approach to select a subset of configuration options to test, which can be dynamically increased in size, depending on the available testing resources.

Qu et al. [143] instead use **static slicing** to determine if, given a configuration, a test needs to be executed under additional configurations. Nguyen et al. [131] dynamically identify which configurations (combinations of configuration values) cover a source code location in order to identify the minimum configurations necessary to cover the highest number of locations using the existing unit tests. Kim et al.'s SPLat [105] requires running a test, then testing all possible combinations of values of the options that were used in that single test, without focusing on other options. Souto et al. [176] extended this approach by considering sampling heuristics like one-disabled and most-enabled-disabled approaches. One-disabled approach consists of testing an application by disabling one option at a time, where most-enabled-disabled consists of testing an application with all its options enabled in a first run and all of them disabled in a second test. Souto et al.'s EvoSPLat [175] focuses on regression testing. EvoSPLat starts by a lightweight analysis that reports under which configuration options a source code modification could be executed, fixes that configuration option's values to guarantee that the modified source code will be covered, then tests the other configuration options' values based on the previously discussed SPLat approach [105].

While most of the work above assumes **that the search space of possible configurations is huge**, and hence needs to be filtered, a separate line of work focuses on **validating this assumption**. Meinicke et al. [118] (using dynamic analysis) and Reisner et al. [153] (using symbolic execution) both found that the interaction between options, i.e., "a partial setting of configuration options such that specific line, block, edge, or condition coverage is guaranteed to occur under that setting, but is not guaranteed by any of its subsets" [153], is surprisingly low. Such low interaction substantially reduces the possible configuration space, and hence makes testing of all relevant configurations easier. To find those relevant configurations, Song et al. [173] proposed an algorithm that uses a low strength covering array, run-time instrumentation, and machine learning to build an interaction tree that represents interactions between options. They extended this approach [174] by considering heuristics related to the interaction between configuration options, which they observed in two open source projects (vsftpd and ngIRCd). Again, they found that interactions (i.e., in terms of code controlled by a combination of options having specific values and without which such code is not reached) are rare.

In contrast to the above work, Keller et al. [101] **evaluate how resilient a software system is to configuration errors** by injecting faults into its configuration files. Xu et al. [213]

extract configuration option constraints from source code in order to report error-prone and inconsistent constraints.

Another direction of testing highly configurable software systems is proposed by Robinson et al. [154], who aims to reduce the number of tests to generate and run by generating tests only for the options changed by a customer compared to the base configuration. Their approach consists of identifying the options modified by the user, finding the impacted area of the source code via control flow and data flow analysis, then selecting or generating tests that cover those source code locations.

Cohen et al. [60] developed a family of greedy Combinatorial Interaction Testing (CIT) sample generation algorithms that aim to reduce the number of tests to run by generating samples that satisfy the software configuration constraints. Comparative evaluation of the cost-effectiveness of these algorithms on four real-world highly-configurable software systems and on a population of synthetic examples shows that their techniques reduce the cost of CIT in the presence of constraints to 30% of the cost of widely-used unconstrained CIT methods, without sacrificing the quality of the solutions.

R9.3 Configuration-aware Reviewing 14 surveyed developers recommend reviewing changes to configuration options. Developers should "*carefully review any pull request that impacts configurations, reviewers must be not only developers but documentation writers and UI experts to make sure that [not only will the changes] just work, but it is clear how to use it and well documented*". This is less obvious than it sounds, since patches impacted by configuration options typically do not show the conditional checks (which are higher up in the code) nor the default option values (which are in separate files or other media). Only if the patch itself manipulates an option, reviewers receive an explicit hint that the patch is impacted by some option(s). Still, they need to manually check if the newly added/removed configuration-related code impacts other code that is not part of the patch. Because of this "hidden" nature of configuration-related code and data, one surveyed expert went as far as proposing to send "*configuration options as a separate patch*". The presence of detailed documentation or comments about configuration options (C5.2) might have a positive impact on configuration-aware reviewing.

R9.4 Fail-safe Cross-Environment Configuration 3 developers believe that configuration options should be easy to adapt to different environments (e.g., testing, staging and production), ideally in an automated fashion. Furthermore, the deployment of the software configuration storage media with the chosen values of configuration options should be defined as a workflow. Following one of the developers, "*nothing is more frustrating than having to*

figure out how to put an environment specific configuration on release day. A CI job which you can just trigger is ideal for deployment".

R9.5 Approaches to Help Users Configure their Software In contrast to R1.4, the techniques discussed here aim to help users customize the default configuration options to their context (instead of finding generic default values that work for as many end users as possible). One of the first approaches to help J2EE developers choose an optimal configuration value is proposed by Raghavachari et al. [149]. It allows to select a range of possible configuration values for each option, select a random value from that range, deploy the web application with that configuration and measure its performance, then choose a second value in order to compare its performance until finding a better configuration. Note that the approaches discussed for value generation in R1.4 also apply to R9.5 [58, 66, 70, 79, 81, 109, 134, 135, 156, 170, 171, 183, 207, 223], but applied by end users instead of developers.

Configuration complexity can be reduced by using Hamidi et al.'s [82] approach to reduce the number of decisions a user has to make in order to configure her software. Their approach models software configuration as a set of options that have relations between them, for example to express that whenever a user decides to assign a value v1 to an option X, she assigns a value v2 to another option Y. Such patterns can be used to automatically change configuration values based on few user decisions.

Huang et al. [86,167] support developers configuring Java frameworks by recommending XML configuration snippets collected from open source repositories. Chen et al. [196] automatically detect correlated configuration options in multi-component/layer architectures by using a database of configuration choices obtained from online fora and websites. Ramachandran et al. [152] instead analyze the names and values of options, complemented by configuration data made available for analysis by software vendors or available online (e.g., fora). Finally, Jin et al. [94] proposed an approach to help users find which options they need to change the value for in a given situation, instead of asking around in an online forum.

While the above approaches for R9.5 seem promising, we did not record any usage of such automated techniques in our interviews or survey.

4.7 Implications

This section discusses potential implications of our findings for both practitioners and researchers.

4.7.1 Implications for Practitioners

While many configuration engineering activities are not covered by academic literature, many challenges related to creation of configuration option, configuration knowledge sharing and quality assurance have been studied. In particular, a substantial amount of work focuses on default value generation (R1.4), configuration failure debugging (R7.3), documentation generation for end users (R8.2), configuration validation (R9.1), configuration-aware testing (R9.2) and optimization of option values for end users (R9.5). While this work is of course not complete, practitioners should look into these techniques, provided they have access to the corresponding academic pulications.

Even if the other configuration engineering activities and challenges saw substantially less academic interest, practitioners can still build on their experience with programming best practices, since many of those still apply in the context of configuration options. In fact, one of the interviewees literally quoted that "*Configuration is code too*". Known programming tenets like the KISS principle (R1.1 and R2.1), option cohesion (R1.3), option granularity (R3.1), encapsulation of sensitive option data (R3.2), good naming conventions (R5.1), choosing the right abstraction for storage media (R2.2) and API for configuration access (R4.1), reuse of third party configuration libraries (R4.2), generation of clear logs (R7.1) and consistently documenting failures and resolutions (R7.2), all still apply to the context of configuration engineering. While standard methodologies and tool support might not exist, organizations could create their own based on these general principles.

In addition to these technical principles, we also identified some organizational recommendations for which academic support is missing. A team should not allow any developer to add or manipulate configurations, but should rely on expert developers with a broad vision of the software system, its architecture and its configuration options (R1.2). Similarly, the organization should establish guidelines to force developers to maintain configuration compatibility (or at least ensure traceability) between different versions of the application (R6.2). Finally, option developers need to know how to effectively communicate with other stakeholders the rationale and constraints of new or changed options (R8.1). For the moment, less guidance exists for these organization-level challenges.

Finally, the main challenges for practitioners in terms of configuration engineering support relate to comprehension of options and values, maintenance of options and quality assurance. In particular, tool and methodological support is needed for configuration understanding (R5.2), option maintenance (R6.1) and configuration-aware reviewing (R9.3). While identified as major challenges that impact the quality and maintainability of an application,

surprisingly little research has been performed on them. At the same time, no general principles exist that could guide practitioners in the short-term.

4.7.2 Implications for Researchers

The first implication that we would mention is for researchers to advertise or publish their work in venues where developers can find them, for example industry conferences or even just simple blog posts. This follows from repeated comments of interviewed and surveyed practitioners about either not being aware of academic results or not having access to them. In particular, the wide variety of approaches related to debugging configuration failures saw only little adoption. While the adoption problem of course applies to any research domain, it especially matters in the domain of configuration engineering, which is at the same time specialized and general-purpose.

As discussed in the previous section, apart from the existing work on recommendations R1.4, R7.3, R8.2, R9.1, R9.2 and R9.5, most of the other recommendations, challenges and activities have been covered substantially less or sometimes not at all. This opens up a wide range of new research opportunities. Here, we point out some obvious opportunities, but many more exist. Basically, any cell in Table 4.3 could be potential avenue for future work.

First of all, it would be interesting to empirically study the impact of the programming best practices discussed earlier in the context of configuration engineering. For example, do naming conventions, reuse of third party configuration libraries and pair-programming improve the quality of software configuration in general? Related to this, many configuration formats exist, including key-value pairs, hierarchical configuration options, or even complete DSLs for configuration. Further guidelines are required to help practitioners decide what format to use for which configuration goal and for which kind of users (technical engineers vs. regular end users).

We found that developers do not plan the evolution of their configuration options, while they fear thorough maintenance of options (i.e., option refactoring and removal). Therefore, developers require approaches to help them detect and remove dead options, or perform option refactoring in general. In practice, developers also need automated approaches to keep the configuration storage media in sync with the options used in the source code, in order to propagate any code changes (e.g., renaming) made to the options. This is a challenging problem, since we found that option names often are not literally mentioned in the source code. This same problem renders the understanding of software configuration in general complex. There are some existing approaches that locate configuration accesses in the source

code, but even they still need to be improved to deal with string concatenation in the name of configuration options.

In general, the existing work on configuration activities, challenges and recommendations typically focuses on individual applications, without considering interactions with the execution environment or other layers of a software stack [162, 164, 197]. Apart from exploring the previously unexplored configuration activities, future work should also revisit existing analyses and methodologies from a system perspective instead of the application perspective. This will allow to develop frameworks and libraries for system-level configuration engineering, including system-level validation, debugging, etc.

4.8 Threats to Validity

Despite the effort spent on our qualitative study design, and gathering and clustering data from three different sources (expert interviews, a large survey, and a systematic literature review), we identified a number of threats to validity.

Internal Validity As first internal threat to validity, our results can miss challenges that are not faced by our interviewees. To mitigate this risk, we invited and interviewed experts that play different roles in different domains. Our 14 interviewed industry experts range from managers to developers and freelancers. They belong to different companies (located in four different countries) and work on different technologies and platforms.

The second threat to validity is related to the data collection during our interviews, as it is possible to misunderstand a challenge or recommendation discussed with interviewed experts. To mitigate this risk, we conducted all our interviews in face-to-face meetings (except one via a skype call), and all interviews were attended by two authors of the paper to avoid any confusing interpretation or misunderstanding. One author led the interview, while the other took notes. At the end of each interview, a debriefing was held to validate and complete the notes.

A third internal threat to validity is related to the data collection in our survey. Surveyed participants might have answered questions incorrectly due to different factors, such as misunderstanding of a question, lack of experience or lack of interest. To mitigate this risk, two authors carefully prepared the questionnaire, while the two other authors reviewed it. During this prepration, we avoided using any research-specific terms. In addition, we performed a pilot study with 9 respondents that allowed us to validate and improve the survey before sending it out to GitHub developers. In particular, the pilot study allowed to reformulate unclear questions or descriptions, restructure the questionnaire to make it easier to

follow, shortening the list of questions by merging related questions, and incorporating other comments.

Furthermore, based on our selection criteria, the invited open source developers all are serious contributors to top Java projects on GitHub, and they were promised that responses would be treated anonymously. Only two unprofessional responses had to be filtered out. Our decision to focus on the top 1,000 most active Java projects allowed to filter out projects that are not software engineering-related or that are too small to actually need configuration options, since their developers would not be knowledgeable about configuration options. Our demographic results for the number of options in the survey respondents' projects confirm that the number of commits was a good proxy for configuration expertise. If one would know the concrete configuration access API used by each GitHub project, one could perform a more accurate sampling based on actual configuration API usage.

The fifth internal threat to validity is related to the difference between interviewed and surveyed participants' contexts (industrial vs. open source projects), and due to which we might miss some challenges or recommendations specific to either context. As is clear from the discussion of our results in sections 4.5 and 4.6, both types of participants agreed on the majority of challenges and recommendations, except for C9.1. For the latter challenge, open source projects rely on GitHub's pull request mechanism to review configuration as well as code changes, while the pull request mechanism could be less adopted in industrial contexts. Furthermore, no additional configuration activities were identified from the survey responses' open questions.

The sixth internal threat to validity concerns our three card sort analyses, which respectively concern the analysis of the interviews, open survey questions, and the classification of the papers obtained from our systematic literature review. Such card sort analyses could lead to subjective results, which could influence our results and findings. To mitigate this risk, all our card sort analyses initially were conducted by two authors for the interviews, and by one author for the survey and systematic literature review. Then, each of these card sort analysis results were reviewed by the other authors. In case of conflicts, the card sort classifier and the reviewing authors together discussed the conflicts to reach a final decision. Finally, the consistent results across all analyses provide further confidence in our findings.

Our seventh internal threat to validity concerns the four SLR selection criteria, since they shaped to a large extent the final selection of papers. In particular, we used criterion 2 to reduce noise in our first iteration caused by systems research on execution environment configuration or tweaking of system performance. The latter domain uses configuration options as a means rather than a goal, and focuses especially on the performance of middleware

and environments. While the results of our first iteration based on the selection criteria might make us miss relevant papers, the recursive snowballing iteration allowed us to recover missing papers and their relevant references, helping us complete a stable set of papers.

External Validity As with all qualitative studies, there is an inherent risk to generalize our findings for all organizations. While the 14 interviewees and 229 respondents cover a wide range of projects and application domains, and provided consistent results across the discussed challenges and recommendations, future work should consider additional organizations and developers. Furthermore, we only considered Java projects in the interviews and survey because (1) it is one of the most popular programming languages and (2) the Java ecosystem has a wealth of configuration frameworks [65, 163]. Other languages like C++, Python or Javascript might experience different configuration challenges and recommendations. Hence, future work should consider such languages. The results of our systematic literature survey are not language-specific.

Despite these limitations, we noticed that the major challenges and recommendations that came out of our study were saturated, in the sense that the progressive card sort analyses on the three data sources confirmed earlier activities and challenges, without suggesting new ones. R7.3 and R9.5 only popped up in the systematic literature review. Further studies on different domains and participants are required to verify our results and make our findings more general.

4.9 Conclusion

This paper aims to improve the general understanding of the software configuration engineering process, its challenges and existing practices (both from practice and academia). Through a series of 14 semi-structured interviews, a survey with 229 GitHub developers, and a systematic literature survey, we found how configuration engineering comprises 9 activities, and is impacted by 22 open challenges. Finally, we discussed 24 recommendations to overcome or avoid the challenges, derived from practitioners' experience and academic literature. The data of our studies is available online [15].

Our interviews identified an initial set of activities, challenges, and recommendations, which were then enriched and confirmed via the survey. The survey also provided insights about the popularity of the challenges from a larger set of participants. On the other hand, our systematic literature review revealed various recommendations that are not fully covered, or not covered at all, by the literature (R1.2, R1.3, R2.1, R4.1, R5.1, R6.1, R6.2, R8.1, R9.3

and R9.4), and two additional recommendations for which we did not record any usage in the practice (R7.3 and R9.5).

These contributions have a number of major implications for practitioners and researchers. First of all, they consolidate the state of the art and state of the practice in configuration engineering, providing both practitioners and researchers a clear overview of this domain, its challenges and best practices. Second, our findings reveal activities and recommendations without substantial research effort thus far, in particular the choice of configuration framework (R4.2), comprehension of configuration options (name, meaning, etc.) (R5.1), dead option detection (R6.1) and configuration-aware reviewing (R9.3). While the other challenges and practices of course have value, these in particular could need more attention.

Finally, our study could be extended outside the scope of run-time configuration options, in particular to pre-deployment configuration engineering (e.g., using conditional compilation), as well as to other roles and software domains.

CHAPTER 5 ARTICLE 2: DOES THE CHOICE OF CONFIGURATION FRAMEWORK MATTER FOR DEVELOPERS?

Mohammed Sayagh, Zhen Dong, Artur Andrzejak, and Bram Adams
Published in the 17th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)

Abstract: Configuration frameworks are routinely used in software systems to change application behavior without recompilation. Selecting a suitable configuration framework among the vast variety of existing choices is a crucial decision for developers, as it can impact project reliability and its maintenance profile. In this paper, we analyze almost 2,000 Java projects on GitHub to investigate the features and properties of 11 major Java configuration frameworks. We analyze the popularity of the frameworks and try to identify links between the maintenance effort involved with the usage of these frameworks and the frameworks' properties. More basic frameworks turn out to be the most popular, but in half of the cases are complemented by more complex frameworks. Furthermore, younger, more active frameworks with more detailed documentation, support for hierarchical configuration models and/or more data formats seem to require more maintenance by client developers.

5.1 Introduction

Modern software systems are expected to be highly configurable to satisfy the personal requirements and preferences of their users. For instance, the distributed storage and computing framework Apache Hadoop 2.7.1. has more than 800 configuration options, where the web browser Mozilla Firefox 43.0 has over 2,000 configuration options available to users! The impact of these options ranges from controlling active features, to adjusting the performance, storing GUI preferences or integrating a software product with its run-time environment (e.g., URL of database or web server). Users typically are able to change the value of configuration options via dedicated configuration files, command line parameters or even at run-time. For example, Firefox has a dedicated "about:config" configuration page to change its configuration on-the-fly.

Since the configuration of a software system is not trivial, but has to support different storage formats, offline/online manipulation, etc., developers are not required to develop their own framework, but have access to a wide range of general-purpose, open source configuration frameworks for their programming language of choice. Quick searches for "configuration" in

Google or StackOverflow yield dozens of hits for each programming language, varying from tiny one-man configuration frameworks (e.g., Raihan's jconfig [151][1]), frameworks included in the standard library of a language (e.g., Java property files) to large, established configuration frameworks (e.g., Typesafe configuration framework [188]). In addition to this, large communities such as Mozilla Foundation or Apache Foundation developed their own configuration frameworks that have become popular outside as well [41].

This paper analyzes whether the choice of configuration framework actually matters from the point of view of developers. Indeed, various studies [80, 127, 133, 146, 213] have shown how software configuration errors are one of the major causes of today's system failures, with sometimes up to 27% of reported issues labeled as configuration-related [215]. A significant part of such configuration errors are caused by improper design and implementation of configuration-related code and configuration options, such as lack of explicit log messages for configuration errors or lack of type checking of configuration values [213], or the constant need for maintaining too strongly coupled configuration-related source code. Such errors could be avoided by choosing the right configuration framework for a project. Furthermore, depending on how a configuration framework is integrated into a project (modular access vs. high coupling), and the rate of new releases of the framework, developers might need to spend substantial effort maintaining their use of the configuration framework.

Hence, this paper performs an empirical study of the characteristics, popularity and maintenance overhead related to the use of 11 general-purpose Java configuration frameworks by analyzing 1,938 GitHub-based Java projects. The contributions of our work are the following:

- Using manual analysis, we build a taxonomy of the major features of configuration frameworks. This taxonomy helps understand the wide range of features and differences amongst the frameworks.
- We study the popularity of the 11 frameworks by analyzing 1,938 projects sampled from GitHub, and which frameworks are typically used together in a project.
- We build classification models to understand the project and framework characteristics impacting the maintenance effort for using a configuration framework.

The results of this paper will help to understand the amount of attention necessary to select a configuration framework for a client project.

[1]Not to be confused with the jConfig framework studied in this paper [90].

5.2 Background and Related Work

This section provides background and related work on software configuration and configuration frameworks.

5.2.1 Software Configuration Frameworks

Software configuration is the mechanism used to adapt a software system to different contexts, simply by changing the value of certain configuration options. As such changes can be performed by end users, no recompilation is required. For example, a user can change the database used by a software system only by changing the URL, username and password in the associated configuration option.

Such options can be stored in different storage formats and can use different configuration models. Common storage formats are JSON or XML files, SQL databases or (more advanced) distributed configuration databases such as ZooKeeper. Configuration models can range from basic key-value pairs to more elaborate hierarchical structures used for example by the Linux kernel or the Windows Registry, where related low-level options are combined into groups that can be combined recursively with other option groups.

Many configuration frameworks have been proposed to manage configuration options, storage formats, models, and other configuration-related functionality. Such frameworks provide an API for developers to read an option's value within a project's source code, such that software developers do not need to implement their own framework. For each programming language, dozens of open-source configuration frameworks are available, each with its own focus and feature set. For example, one of the simplest Java configuration frameworks is *Java Properties*, which allow to read key-value configuration pairs from textual files. However, many other, 3rd party frameworks exist. In this paper we study their differences and impact on software maintenance.

5.2.2 Related Work

As configuration issues contribute 17% of the total cost of ownership of today's software, troubleshooting misconfigurations is a large part of technical support [99]. Given the role of configuration frameworks in avoiding configuration errors, it is important to understand the impact of configuration framework choice for a software project. Several researchers have focused on analyzing and comparing configuration frameworks, although without considering the impact on developers. Moreover, a large body of research exists on configuration errors and debugging.

Configuration Frameworks. Rabkin *et al.* analyze configuration frameworks in 7 large-scale Java software projects and find that all 7 projects used a standard key-value configuration model to organize configuration data. They also made a taxonomy of configuration options, concluding that most configuration options fall into a small number of data types. Differently, our work focuses on the taxonomy of existing configuration frameworks and their properties when used in the application development.

Other research groups studied projects that use a hierarchical configuration model, in which options are organized into a tree instead of a flat key-value model [50, 95]. Such hierarchical configuration data is widely used [95]. Tresch investigated common configuration frameworks for Java, briefly summarizing the application scenarios and configuration data formats for each framework [187], while Denisov summarized the features of three major configuration frameworks (java.util.prefs.Preferences, java.util.Properties and Apache Common Configuration) [65]. Instead of focusing on a handful of configuration frameworks and some of their features, we performed a detailed analysis on 11 configuration frameworks for Java.

Configuration Errors. Xu *et el.* [212] study 620 user-reported configuration errors from 4 software projects, i.e., Apache HTTP server, MySQL database, Apache Hadoop and a commercial storage system. All these projects have the over-designed configuration problem, i.e., they offer hundreds of configurations that are unused, but may impact behavior of the system in unforeseen ways if touched incorrectly. For instance, 51.9% options can be removed without impacting the usage of the system. Yin *et al.* [215] also study 546 misconfigurations from four widely used open source systems (CentOS, MySQL, Apache Http Server, and OpenLDAP). Of these, 70% to 85.5% are due to mistakes in choosing the value of configuration options, while another significant number of misconfigurations are due to compatibility issues between different components or modules. Those misconfigurations can be reduced by adopting a well-designed configuration mechanism in software development.

Arshad *et al.* [43] study 281 bug reports related to configuration for the GlassFish and JBoss Java EE application servers. They find that a significant part of configuration errors are due to mistakes by the developers and require code modifications to fix the problem. Sayagh *et al.* [162, 164] studied configuration errors that span across different layers of a software stack such as LAMP. One of the causes of such errors is the diversity of configuration frameworks and models used across layers. Other research [68, 69, 147, 218] has studied the impact of configuration models such as key-value pairs on software misconfiguration. All this work indicates that configuration frameworks play an important role in configuration errors and maintenance, and hence requires careful selection. This paper analyzes this role.

5.3 Taxonomy of Configuration Frameworks

In this section, we introduce the 11 Java configuration frameworks that we are studying, as well as the taxonomy of their features and properties that we derived.

5.3.1 Configuration Frameworks

This paper focuses on general-purpose configuration frameworks, i.e., configuration frameworks that can be used in a variety of software systems, from desktop applications to mobile apps or enterprise software. As such, our analysis is relevant to a wide range of systems. Furthermore, given Java's 20+ years of history and the many (open source) configuration frameworks available for it, we focused exclusively on general-purpose Java configuration frameworks. Other programming languages left for future work.

To obtain the catalog of Java configuration frameworks used in this paper, the first two authors performed search queries using different keywords and phrases such as "Java configuration frameworks" and "Java configuration tools", then read a large amount of technical fora and blogs. They quickly converged on a set of 14 frameworks covering a wide range of mature and young frameworks, which are shown in Table 5.1, ordered based on the date of their first release.

Before considering the properties of these frameworks in more detail, it is important to note that some configuration frameworks were not included in the paper. Play [140] is a web application framework whose configuration framework basically is a modified version of the Typesafe framework. Furthermore, neither Carbon [55], nor Raihan's jconfig [151] had any GitHub project as user, hence we excluded these frameworks as well. Android's SharedPreferences framework [165] was excluded, since it only applies to Android apps, and not to desktop or enterprise applications. Finally, we did include Spring and Deltaspike, since they support desktop and enterprise applications, although they do not support mobile apps. The resulting set of 11 frameworks is used in the remainder of this paper.

5.3.2 Taxonomy

In order to understand the differences between configuration frameworks in terms of features and properties, and later relate those to the popularity and maintenance effort involved with the usage of these frameworks, we built a taxonomy of configuration frameworks, then classified each framework according to the taxonomy.

Table 5.1 The studied configuration frameworks and their signatures.

#	Framework	Released	Signature
1	Properties	01.1996	import java.util.Properties
2	System	01.1996	System.getProperty
3	jConfig [90]	10.2001	import org.jconfig.*
4	Preferences	07.2003	import java.util.prefs.*
5	Spring [177]	06.2003	import org.springframework.*
6	Commons [41]	11.2005	import org.apache.commons.configuration.*
7	Constretto [61]	05.2010	import org.constretto.*
8	Typesafe [188]	12.2011	import com.typesafe.config.*
9	Deltaspike [64]	10.2012	import org.apache.deltaspike.*
10	Owner [137]	12.2012	import org.aeonbits.owner.*
11	CFG4J [57]	07.2015	import org.cfg4j.*

To determine the taxonomy's properties, for each framework at least two of the authors manually studied the public documentation and browsed forums and blog posts associated with each framework. Any relevant property recurring within the analyzed framework was tagged. Then, once all frameworks were analyzed, we compared the tagged properties across all frameworks to arrive at a final list of 17 framework properties, grouped into 3 dimensions. The resulting taxonomy, as well as each framework's classification, is shown in Table 5.2. Note that each framework's tagged properties were checked by two authors, while the full set of properties was obtained by all authors together.

The first taxonomy dimension, i.e., *General Properties*, contains basic information about the configuration frameworks. *Universal* indicates whether a configuration framework is fully general-purpose, or does not support mobile apps. *JDK-Standard* indicates if a framework is integrated into the Java SDK libraries. *Age* is *low* if a framework was created after 2010, *high* if before 2000, *med* if in between 2010 and 2000. If there is no documentation for a framework, we consider its *Quality of Documentation* as *low*. If the documentation is not very comprehensive and/or written in a non-rigorous manner, the quality is considered as *med*. Finally, for comprehensive and concrete documentation, we consider the quality as *high*. To mitigate subjectivity of assessing quality of documentation, for each framework, two persons separately looked for related documents such as JavaDoc and tutorial documents, and had a discussion to decide its value. *Actively Maintained* indicates whether there has been at least one commit to the configuration framework's code repository in 2016.

The second dimension, i.e., *Feature Richness*, measures how powerful the framework is. *Multiple Storage Formats* measures the number of data formats (such as XML, properties files or JSON) a framework supports: *low* (1∼2 formats), *med* (3∼4 formats) and *high* (≥5 for-

mats). *Hierarchical Configuration Structure* indicates if a framework supports hierarchically organized configuration data (e.g., tree structured) or just flat key-value pairs. *Hierarchical Overriding* means that the value of an option configured in a lower priority layer can be overridden by a higher priority layer. *Multiple Data Source* indicates that a framework is able to load configuration data from multiple sources instead of just from one file. *Variable Substitution* specifies whether the user can define and use variables in the configuration file instead of having to copy repetitive configuration values throughout. *#API methods* is the number of public methods within public classes of each configuration framework. Similarly, *#Annotations* is the number of public Java annotations proposed by each of the studied configuration frameworks. The last two metrics are basically obtained by using the JavaParser tool. Note that for all these frameworks, methods that are within test classes are ignored. In addition, since Deltaspike and Spring contain more than just configuration functionality, we consider for these two frameworks only the classes whose own name or package name contains the keyword *config*.

The final dimension, *Programming Support*, contains properties supporting programmers when integrating the configuration framework in their source code. *Dependencies* measures how many dependent libraries need to be imported before using a configuration framework: *none* (0), *med* (0∼10), and *high* (≥10). *Distributed Environment Support* indicates if a framework can be used in a distributed setting, for example through the use of a configuration database. *Type-safety* indicates that a framework checks whether the value of a configuration option has the right type (e.g., double vs. integer) when read. *Notification Mechanisms* specifies that a system will get a notification from the configuration framework when configuration data has been changed by the user. Finally, *Configuration Injection* allows configuration to be fully defined in external files, with the corresponding configuration values automatically injected in the source code.

Apart from the three frameworks included in the SDK (Properties, System and Preferences), all frameworks cover a wide range of features, which confirms the need for a study like this paper! Only Type-safety is shared amongst all non-SDK frameworks, but no other clear pattern of feature usage can be found. The most rare features are Distributed Environment Support and Notification Mechanisms, with jConfig, Spring and Commons supporting both of these. The latter two frameworks are the most fully featured, followed by Owner and jConfig.

5.4 Collected Data

Now that we understand the different features of the configuration frameworks, we aim to study the popularity of each framework (Section 5.5) as well as any relation between framework or project features and the amount of maintenance effort required in projects using those frameworks (Section 5.6). Each of these analyses uses a different data set of GitHub projects, which we will refer to as "Data Set 1" (popularity) and "Data Set 2" (maintenance effort).

5.4.1 Data Set 1: Popularity

Sampling. We first used GHTorrent to create a list of all non-fork Java projects on GitHub with at least 50 commits. The latter constraint is important to eliminate as many toy projects as possible, or repositories that are just mirrors (and hence only have few GitHub commits) [98]. Then, we randomly sampled and cloned 10,000 projects from the list. To verify the diversity of this sample, we used the approach of Nagappan et al. [126] to compute a *diversity score*. Our diversity score considered the following metrics related to project maturity: number of commits, number of authors, number of committers, and active age (time span between first and last commit). The overall diversity score of 1.00 indicates that our sample is sufficiently diverse.

Despite the high diversity score, the sample still contained many repositories not used for developing software projects [98], such as experimental repositories and repositories containing example code snippets, demos, personal test code, and so forth. Since those repositories are not used to develop software projects, we tried to exclude them from the sample, by selecting only repositories that have at least 3 stars. This yielded Data Set 1, which contains 1,938 repositories (Table 5.3). Note that the number of stars was not available from the GHTorrent database, and hence could only be obtained by scraping the projects' GitHub repositories (using Christophe et al.'s crawler [59]).

Mapping projects to configuration frameworks. To learn which configuration framework(s) is/are used by a project, we automatically scan the projects' source code for *framework signatures*. These are statements indicating that the particular configuration framework is used in the code, see Table 5.1. For most frameworks, the signature is a set of import statements that we manually extracted from the javadoc documentation of the corresponding framework. Only for System Properties the signature is not an import statement (since no import is necessary to use this framework), but instead we search directly for the *System.getProperty*()

calls. After scanning the projects of Data Set 1 for these signatures, for each project we end up with zero or more corresponding frameworks being used in the latest snapshot.

5.4.2 Data Set 2: Maintenance Overhead

To study maintenance overhead involved with configuration framework usage, we cannot use Data Set 1, since due to the sampling used we might not have sufficient projects for each configuration framework. Instead of random sampling, for Data Set 2 we used the frameworks' signatures to search for projects using each framework in turn.

Querying. To select Java projects that use at least one of the 11 Java configuration frameworks, we used the GitHub search function with a configuration framework signature as a search keyword, for one framework at a time. Since GitHub's web search is limited to 100 pages of search results, each containing 10 files, for each framework we obtained a maximum of 1,000 pages matching the framework's signature. To increase this number, we performed each search three times, since GitHub offers three different ranking algorithms for its search results (*"Best Match, Recently indexed, and Least recently indexed"*). Only Constretto and CFG4J yielded less matches than the maximum number of search results.

Using the GitHub search webscraper of Christophe *et al.* [59], we obtained the GitHub project names mentioned on the 100 pages of search results, and after removing projects that are forks, we obtained the projects summarized in Table 5.4. We scraped their GitHub pages for repository-related meta-data (like number of releases and stars), and also cloned them to analyze their code change history.

Mapping projects to configuration frameworks. We used the same approach as for Data Set 1 to identify the configuration frameworks used by each project in Data Set 2.

5.5 Popularity of Configuration Frameworks

In this section, we study the popularity of configuration frameworks in Github Java projects by addressing two research questions.

RQ1: How Popular are Individual Configuration Frameworks?

Motivation. Although Table 5.2 contains a wide variety of publicly available configuration frameworks, many of them have comparable features, for example support for multiple formats of persistent storage, variable substitution, or type safety. Consequently, selecting a configuration framework suitable for a specific Java project typically requires a time-

consuming evaluation of alternatives. By assuming that popularity of a framework might be an indication of its maturity and quality, a study of configuration framework popularity can provide hints for developers about which configuration frameworks (and hence features) to prefer.

Approach. For each project in Dataset 1, we identify which configuration framework (or frameworks) it was using at the time of writing this paper. From this information, we compute the popularity statistics of Table 5.3. In particular, we calculate two metrics to measure the popularity of each framework: the number of projects using this configuration framework, and the number of projects *only* using this configuration framework at the time of writing this paper.

Results. Out of the 1,938 projects of Dataset 1, 1,034 projects use one or more configuration frameworks. The popularity of each framework is shown in Table 5.3.

Finding 1: System Properties and (java.util) Properties are the most widely used frameworks. There are 821 projects in the popularity dataset that use the System Properties framework. Similarly, the Properties framework is widely used as well, namely by 600 projects. This result of course comes as no surprise, since these configuration frameworks are integrated in the Java SDK and hence can be directly used without importing any external libraries. However, as we could see in Table 5.2, these are also the weakest frameworks in terms of features.

Finding 2: Third-party configuration frameworks are not that commonly used. Table 5.3 shows that the top three third-party frameworks are Spring, Commons, and Typesafe with 91, 37 and 14 projects using them, respectively (Preferences is also included in the Java SDK). Given that Dataset 1 comprises 1,938 projects, the proportion of projects using third-party configuration frameworks is surprisingly low, below 5%. In fact, for some of the third-party frameworks we did not find any project using it in Dataset 1 (which is why we are using the framework-specific Dataset 2 later in this paper). This is partly due to the younger age of those frameworks.

Finding 3: Among the 904 projects without a configuration framework, 362 projects are Android projects. We found that within the considered 904 projects without a framework there are 362 (40.1%) Android projects. Since the Android platform provides a SharedPreferences framework for storing key-value pairs as well as other mechanisms such as support for SQLite databases and XML files, 209 out of 362 Android projects use the SharedPreferences framework, refraining from using a general-purpose configuration framework. The latter statement is confirmed by the fact that among all 470 Android projects in

the whole sample (of 1,938 projects), only 108 or 23% use one of the studied configuration frameworks

Of the remaining 542, i.e., non-Android, projects without a configuration framework, we randomly sampled 53 projects (10%) for manual analysis. We found that 26 projects in this sample do not use any configuration mechanism. Among these, there were 7 projects containing code examples (e.g., for interviews or exercises), 6 libraries, 4 plugins, 4 small games, and 4 simple applications. Another 27 projects in the sample use either an ad hoc configuration mechanism (mostly XML files), or are plugins for other applications having their own configuration mechanism. Only 17% of these 27 projects are stand-alone applications.

Finding 4: Developers prefer easy-to-use and simple configuration frameworks with good documentation, especially Spring, Typesafe, Commons and Properties. We contacted Java developers via 2 Reddit and 4 Facebook groups with a small survey [180] to better understand the criteria considered by developers to choose a suitable configuration framework (RQ1).

From the 10 replies that we received, we learnt that ease-of-use is the developers' primary criterion for framework selection (4 votes), followed by the simplicity (2 votes), quality of documentation (2 votes) and capability of hierarchical overrides (2 votes). In terms of framework recommendations, Spring was rated highest (4 votes), followed by Typesafe, Commons, and Properties (3 votes each). This explains why standard frameworks like Properties are popular, a finding that was confirmed even more by the following quote: *"Think of building software as digging a hole, and the JDK is your shovel. If you are just digging a hole to plant a tree, your shovel will do fine. If you are digging a swimming pool, then you will want to bring in some heavy machinery (aka 3rd party libraries)"*.

RQ2: How Often are Configuration Frameworks Used Together?

Motivation. We found that some projects are using multiple configuration frameworks at the same time. Given the different focus of frameworks in terms of features (see Table 5.2), this could suggest that some frameworks are complementary and serve different purposes. This RQ aims to understand how common such co-occurrences are as well as which frameworks co-occur often.

Approach. Using Dataset 1, we count how many projects use k configuration frameworks in their most recent Git snapshot), for $k = 1, \ldots, 5$. We also investigate whether there is a relationship between project maturity and the number of configuration frameworks a project uses, since older projects might be larger and hence have heavier demands for configuration

frameworks. We measure maturity in terms of different metrics, such as age of the project, number of authors, number of commits, number of committers and number of source code files.

Finding 5: 47.5% of the projects using a configuration framework (491 out of 1,034) combines multiple frameworks. 52.5% (543 out of 1,034) of the projects use one framework, with 38.5% (389 out of 1,034) using two configuration frameworks. There are substantially less projects with 3 or more configuration frameworks: 8.3% (86 out of 1,034) with 3 frameworks, and less than 0.6% with 4 or 5 frameworks (6 and 1 projects, respectively).

Finding 6: System and Properties co-occur the most with other frameworks. The co-occurrence heatmap in Figure 5.1 confirms that combinations of System with other frameworks dominate the co-occurrence relations, followed by Properties and (at some distance) Spring, Preferences and Commons. This ranking matches with the popularity numbers in Table 5.3, where more popular frameworks co-occur more often with other frameworks.

As mentioned before, System and Properties only provide a limited data set, specializing respectively in access to shell environment variables and to flat files with key-value pairs. For System, no explicit imports are needed and option values can be set via the Java command line, lowering the barrier for using it. This positions both configuration frameworks as an easy-to-use complementary configuration mechanism.

Finding 7: More mature projects tend to use multiple configuration frameworks. The beanplots[2] in Figure 5.2 show that the number of frameworks used by a project is higher for larger projects (there are only few projects with 4 or more configuration frameworks, so these beans are not significant). Other metrics such as active project age confirmed this finding, which confirms our earlier hypothesis about older projects. We could not find any more complex pattern explaining co-occurrence of configuration frameworks.

Finding 8: The surveyed developers use multiple configuration frameworks in the same project because their library dependencies needed them, or because of complementary features provided by configuration frameworks. Only 4 of the developers in the survey were familiar with projects using multiple configuration frameworks. One of the main explanations for co-occurrence of configuration frameworks was dependence on library or component, for example, one developer would add an additional configuration framework *"only if required by a library/framework"*. Another common reason was to exploit the different functionalities of each configuration framework, prompting people to just add a new configuration framework that offers the required features, *"No need to reinvent the wheel for all things. Just add what is missing on top"*.

[2] A beanplot is a boxplot that also shows the density of the data instead of just a rectangle.

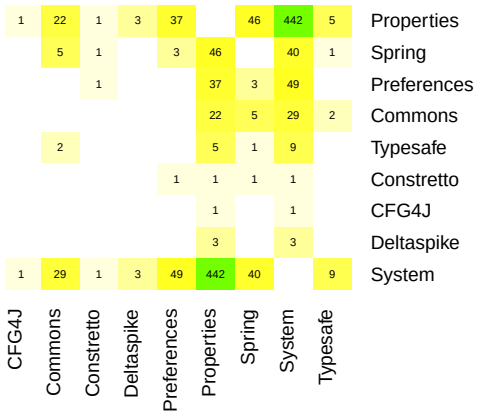


Figure 5.1 Heatmap of co-occurrence of configuration frameworks in the projects using a configuration framework.

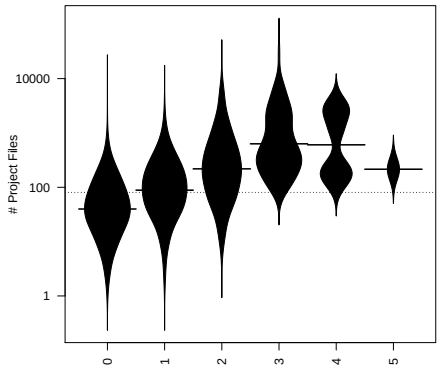


Figure 5.2 Bean plots of the number of files (y-axis) within projects, grouped by the number of configuration frameworks used by projects (x-axis).

5.6 Maintenance Overhead of Frameworks

In previous sections, we compared different Java configuration frameworks in terms of features and studied their popularity across large and popular Github Java projects. We also qualitatively found that people tend to choose easy-to-use and simple configuration frameworks. In this section, we identify the taxonomy and software project-related properties that define such simplicity, in terms of effort required by client developers to maintain a configuration framework.Those properties are the ones a client developer should consider in order to choose a suitable configuration framework. Our findings could also help configuration framework developers to improve their frameworks by focusing on more critical attributes from a maintenance point of view.

We address the following research questions:

RQ3 *Which factors impact the percentage of configuration-related commits?*

RQ4 *Which factors impact the percentage of configuration-related source files?*

RQ5 *Which factors impact the percentage of developers touching configuration code?*

5.6.1 Case Study Setup

The goal of RQ3 to RQ5 is to compare configuration frameworks (in terms of their features) based on the effort required by a developer to maintain them. Maintenance here refers to any changes involving the configuration framework API that a developer needs to do while evolving his or her own software project, for example to update the code to a new version of the framework or to spread configuration data throughout the application.

To perform our analysis, we build classification models explaining one of three effort measures in terms of configuration framework (Table 5.2) and project-related properties. First, we discuss the dependent and independent variables of the models, before discussing how we built and evaluated the models.

Dependent Variables. Table 5.5 shows the three dependent variables (CFCommits, CFAuthors and CFFiles), together with three auxiliary variables they are based on. Each dependent variable measures some facet of maintenance effort. As we are building classification models, the percentages of CFCommits, CFAuthors and CFFiles are discretized into a binary value, using the median as threshold. So, all projects whose percentage of configuration framework-related commits is higher than the median of that percentage across all studied projects are considered as having a "high" value, while others have a "low" value.

To determine when a configuration framework is added (*CFAddedIn*) and removed (*CFRemovedIn*), we identified the commits that add or remove signature instances of a configuration framework throughout a project's git repository. By counting the number of "live" signature instances (+1 for each added instance, -1 for each removed one), we can determine the full removal of a framework if the count hits zero.

However, finding all configuration-related commits, i.e., not just those adding or removing the signature (import) of a framework, is less straightforward. One way to find these commits is to analyse each modified line of a commit for calls to one of the methods of a configuration framework API. However, this approach is not accurate, because the number of methods of a configuration framework can be very large, while configuration framework API methods do not always have unique names.

Therefore, we used a technique similar to the approach used by Zhang et al. [220], which consists of selecting as "configuration commits" those that contain in their commit message one of the following keywords related to configuration: *config, property, properti, pref,*

option, and *setting*. Hence, for each file containing an import (signature) statement of a configuration framework, we consider its "configuration commits" as the commits related to that configuration framework that we need to analyze. We can then calculate the variable *CFCommits*, while the number of unique configuration framework authors is used for the *CFAuthors* variable.

From the same historical data set, we know all the files that contained code calling the configuration framework, across the whole history of each repository. This enabled us to calculate the total number of these source files across time as the variable *CFFiles*.

Project-related Variables. A first set of independent variables is formed by project-related variables, i.e., variables that control for the activity and state of a source code project as a whole. They are shown at the bottom of Table 5.5. We obtained the set of metrics *Watch, Star, Branch, Releases* by scraping each Github project's web page, then used the git repository's logs to extract the metrics related to the number of files, authors, and commits, which are respectively *NbreFiles, NbreAuthors,* and *NbreCommits*.

Configuration Framework Variables. These variables are the 17 configuration framework properties of Table 5.2. Note that all projects using the same framework share the same value for these properties, only their project-related variables and dependent variables differ. Conversely, projects that at the time of writing this paper use multiple configuration frameworks, were included once for each configuration framework, in which case these data instances had overlapping project-related metrics, but different configuration framework properties.

Building the logistic regression models. To build our logistic regression models [97], our initial training set consists of all projects in Dataset 2 (Table 5.4). However, since the values of the configuration framework properties are repeated across all projects using a given framework, this might introduce bias towards the most common configuration frameworks, yielding models that are overfitted on System and Properties. To counter this, we resampled the training set to obtain the same number of instances for each configuration framework. Based on the number of non-fork projects in Dataset 2, we chose 30 projects per framework, which required us to resample jConfig, Constretto and CFG4J with replacement (i.e., duplicating some of their rows) to obtain 30 data points. The resulting resampled data set forms our training set.

We used VIF (Variance Inflation Factor) analysis to remove highly correlated variables (VIF > 5 [97]), then built an initial baseline model only containing the project-related variables as independent variables. We then incrementally add the three configuration framework dimensions of the taxonomy to evaluate the degree to which each dimension adds new information

related to the maintenance effort-related dependent variable (we build separate models for each of the three maintenance effort measures). We use ANOVA tests and AIC score to evaluate how significantly a new model differs to an earlier model. We then calculate precision, recall and AUC values (via 10-fold cross-validation) to evaluate how well the models fit the data in terms of lack of false alarms, ability to find all known high maintenance projects and performance improvement compared to a random model, respectively. We also build a final model with only the statistically significant variables.

Finally, to understand which variables have the highest impact on the dependent variable as well as the direction (increasing/decreasing) of this impact, we used the effect size score of Shihab et al. [168]. It requires evaluating a classification model using the median value of each variable as input, then, one at a time, adding one standard deviation to the median value of a variable (while keeping the other attributes at their median value). For example, if the model output is 50% when all independent variables are at their median value, while the output after adding a standard deviation to the first independent variable is 100%, we say that the latter independent variable has an effect score of $\frac{100\%-50\%}{50\%} = 1$, indicating a 100% increase in probability compared to the baseline effect size. Whereas one cannot directly compare the coefficients of the logistic regression models, as not all variables use the same unit [97], the effect sizes of each variable can be directly compared to each other.

For boolean variables, we used "false" as the reference value for the effect size, and used the value "true" instead of the "median value + standard deviation". Hence, the effect size expresses the effect of moving from "false" to "true".

RQ3. ***Which factors impact the percentage of configuration-related commits?***

Finding 9: The taxonomy variables have a higher impact on maintenance effort in terms of number of commits.

Table 5.6 shows the significant variables in the logistic regression model, split up in project-related and configuration framework-related. In both groups, variables are ordered based on the absolute value of the effect size, from highest effect (either positive or negative) to lowest. It is easy to see how the three project-related variables have only a tiny effect size, close to zero, while the taxonomy-related variables have an effect size of at least 69.1% (negative; QualDocMed).

We find that **the more active the developers of the configuration framework are (ActMaint), less choice of storage formats for configuration files (PersisVariety-Low), or younger configuration frameworks (AgeLow), the more maintenance**

effort the client developers of the framework seem to perform. On the other hand, a **lower quality of documentation (medium as opposed to high; QualDocMed) seem to be linked to lower effort**.

Surprisingly, we found that the higher the quality of a configuration framework's documentation, the more commits the configuration framework requires. This could be explained by the fact that frameworks with *more comprehensive* documentation might be more feature-rich, or that developers look for easy-to-use configuration frameworks (Section 5.5), which tend to have concise documentation. More research is needed to evaluate the needs for good configuration framework documentation.

RQ4. ***Which factors impact the percentage of configuration-related source files?***

Finding 10: Taxonomy variables again have the strongest link with maintenance effort.

We obtain similar findings as for RQ3 (see Table 5.7), with ActMaint, AgeMed (replacing AgeLow) and QualDocMed again figuring amongst the most highly impacting variables, although the sign of the effect size has changed. Furthermore, ScopeUniv and HierStruct are additional impactful variables.

We find that **younger (AgeMed, as compared to AgeHigh) and actively maintained (ActMaint) configuration frameworks, with less documentation (QualDocMed) and a universal scope (ScopeUniv), are spread across less files** in a given source project. Frameworks that support **hierarchical configuration models (HierStruct) are spread across more files**, although this effect size is close to zero.

The finding for AgeMed is as expected, since older frameworks have been used longer by projects, leading to tighter integration. Moreover, this finding can give an indication that developers progressively add configuration frameworks in different source code files, which leads to stronger coupling to the configuration framework.

The scope of configuration frameworks is the third most important factor, indicating that **less general-purpose frameworks like Spring have a stronger coupling** to a software system, likely because they come with more heavy configuration (and other) machinery. The HierStruct finding seems to suggest that hierarchical configuration models have a stronger coupling with source code projects, i.e., require more complicated interactions in a project.

RQ5. ***Which factors impact the percentage of developers touching configuration code?***

Finding 11: Taxonomy variables have the highest impact.

Since RQ5 yields **similar results as RQ3** (Table 5.8), we only discuss the differences. The age of the configuration frameworks does not play a major role, and is replaced by the variables JDKStandard and HierStruct. In particular, **Java SDK frameworks have less developers making changes** to configuration-related code (likely due to their simplicity), while the use of **hierarchical configuration models sees less developers do changes** (in more files, see RQ4).

5.6.2 Discussion

Based on the three explanatory models, we conclude that configuration framework taxonomy metrics have important relations with the effort required to maintain configuration frameworks in a given software project. Surprisingly, none of the significant configuration framework variables belongs to the Programming Support dimension (Table 5.2), while all 5 General Properties and 2 out of 7 Feature Richness properties were significant in at least one model.

One of the two most commonly impactful General Properties is the degree of active development of a configuration framework. The more active, the more commits and developers need to be involved to maintain a client project, although slightly less files in the project actually use the framework. This basically empirically confirms the common knowledge that code reuse implies staying on the lookout for updates.

The other commonly impactful General Property is the quality of documentation. Higher quality involves more maintenance commits by more developers across more files. This was the most surprising finding. We believe that a correct interpretation of this finding is that documentation should be *"simple but not simplistic"*. It should emphasize concise examples about how to integrate a configuration framework, as developers generally have a lack of time to deeply learn a configuration framework.

Older, stable frameworks require less maintenance commits (likely because the framework is less active), but their usage typically is spread across more files in a project. Finally, hierarchical configuration models were associated with less developers making commits, but those commits touched more files of a project.

5.7 Threats to Validity

Regarding threats to external validity, we only considered general-purpose configuration frameworks for Java projects. However, according to Nagappan et al.'s diversity measures [126] (value close to 1), our sample of repositories is representative for the population of Java projects in GitHub. In future work, we plan to investigate other programming languages, as well as domain-specific configuration frameworks like SharedPreferences.

Regarding construct validity, we use the percentage of changes, files or authors touching configuration framework usage as proxies for maintenance effort. Although these are typical approximations for maintenance effort case studies [206], they are still proxies. Although we studied three different proxies, using other metrics should be considered in future work. Note that we did not distinguish bug fix commits from commits adding new features or using new framework APIs, as those would require additional heuristics for analyzing our data on top of the ones we use to identify configuration commits. However, our models do control for the total activity in a project, hence we believe this threat is addressed in a reasonable manner.

Finally, regarding internal validity, we conducted a short survey for which we received 10 answers. We plan to conduct a larger survey to investigate configuration frameworks and their impact on configuration errors and maintenance effort.

5.8 Conclusions

We conducted an empirical study on 11 major Java configuration frameworks in almost 2,000 GitHub projects. A proposed taxonomy of framework features provides evidence of the wide variety of features offered by configuration frameworks. It also supports practitioners in selecting frameworks whose features are most suitable to the projects.

We found that SDK frameworks are the most commonly used frameworks, in half of the cases complemented by more powerful frameworks. Simple frameworks are typically preferred. Also, the choice of configuration framework explains to a large degree the maintenance effort required for configuration, whereas project-specific characteristics play less of a role. More active frameworks with higher quality documentation that have more flexible configuration storage formats have a higher tendency towards more configuration-related changes by more developers. On the other hand, older, less active configuration frameworks with high documentation tend to be used across more files in a software project.

Our findings empirically confirm common assumptions about configuration frameworks (e.g., reuse requires more maintenance if reused framework is active). However, they also raise

new questions about the role of configuration framework documentation or the perils of hierarchical configuration models (stronger coupling), which should be addressed in future work.

Table 5.2 Taxonomy of the 11 studied configuration frameworks (numbered according to Table 5.1).

Dimensions	Properties	1	2	3	4	5	6	7	8	9	10	11
General Properties	Universal	✓	✓	✓	✓		✓	✓	✓		✓	✓
	Part of SDK	✓	✓		✓							
	Age	high	high	med	med	med	med	low	low	low	low	low
	Quality of Documentation	high	high	low	high	high	high	med	high	med	high	med
	Actively Maintained	✓	✓		✓	✓	✓		✓	✓	✓	✓
Feature Richness	Multiple Storage Formats	low	low	med	med	med	high	high	med	med	low	med
	Hierarchical Configuration Struct.			✓		✓	✓	✓	✓	✓	✓	
	Hierarchical Overriding			✓		✓	✓	✓	✓	✓	✓	
	Multiple Data Sources			✓	✓	✓	✓		✓			✓
	Variable Substitution					✓	✓		✓	✓	✓	
	#API methods	15	5	273	39	1,681	1,022	191	109	44	37	57
	#Annotations	0	0	0	0	242	0	6	1	16	12	0
Programming Support	Dependencies	none	none	med	none	high	high	med	none	none	none	med
	Distributed Environment Support			✓		✓	✓					✓
	Type-safety			✓	✓	✓	✓	✓	✓	✓	✓	✓
	Notification Mechanisms			✓		✓	✓				✓	
	Configuration Injection					✓	✓	✓		✓	✓	

Table 5.3 Popularity of configuration frameworks in Dataset 1. Columns "#/ % Projects" report the number/percentage of projects using a given framework (projects may use multiple frameworks), while column "# 1 CF" shows the number of projects using *only* the given framework.

Framework	# Projects	% Projects	# 1 CF
System	821	42.36	347
Properties	600	30.96	142
Spring	91	4.70	38
Preferences	57	2.94	6
Commons	37	1.91	5
Typesafe	14	0.72	4
Deltaspike	3	0.15	0
CFG4J	1	0.05	0
Constretto	1	0.05	0
jConfig	0	0	0
Owner	0	0	0
Any Framework	**1,034**	**-**	**542**

Table 5.4 #Projects in Dataset 2.

CF	# Repositories	# Non-fork
System	889	338
Commons	836	442
Spring	754	291
Properties	744	399
Preferences	662	408
Typesafe	659	383
Deltaspike	232	157
Owner	144	82
jConfig	53	17
Constretto	37	24
CFG4J	13	7
Total	**5,216**	**2,575**

Table 5.5 Auxiliary, dependent and project-related (control) variables considered in the maintainability models.

Category	Metrics	Description
Auxiliary Variables	CFAddedIn	Date when a configuration framework is added in a project.
	CFRemovedIn	Date when a configuration framework is removed from a project.
	TotalCommitsInCFPeriod	#Commits between the adoption and removal date of a framework in a given project.
Dependent Variables	CFCommits	Percentage of commits touching files that access a given configuration framework.
	CFAuthors	Percentage of authors that change files that access a given configuration framework.
	CFFiles	Percentage of source code files of a project that access a configuration framework.
Project-related (control)	Watch	Number of watches of a Github project.
	Star	Number of stars of a Github project.
	Branch	Number of branches of a GitHub project.
	Releases	Number of releases of a Github project.
	NbreFiles	Number of files within a GitHub project.
	NbreAuthors	Number of authors of a GitHub project.
	NbreCommits	Total number of commits in a GitHub project.

Table 5.6 Model for RQ3 (AIC: 355.63, Prec.: 77.41%, Recall: 72.72%).

Attribute	Coefficient	Std. Error	Signif. code	Effect size
NbreAuthors	0.009478	0.002097	***	1.72494E-05
Releases	0.001029	0.0002324	***	2.07543E-07
NbreCommits	-0.0002525	0.00005047	***	-1.10599E-08
ActMaint	1.679	0.3764	***	2.400912759
PersisVarietyLow	1.214	0.34	***	1.564812448
AgeLow	1.167	0.3019	***	1.485730827
QualDocMed	-1.277	0.506	*	-0.691757625

Table 5.7 Model for RQ4 (AIC: 331.87, Prec.: 74.5%, Recall: 90.30%).

Attribute	Coefficient	Std. Error	Signif. code	Effect size
Releases	-0.003724	0.002345		-2.90491E-08
NbreFiles	-0.0006832	0.0001462	***	-3.32257E-10
NbreCommits	0.00007836	0.00003293	*	8.5836E-12
ScopeUniv	-3.797	0.7478	***	-0.12657923
QualDocMed	-2.841	0.8845	**	-0.050931693
AgeMed	-2.203	0.489	***	-0.026086136
ActMaint	-1.825	0.4438	***	-0.017012309
HierStruct	0.8339	0.367	*	0.001885136

Table 5.8 Model for RQ5 (AIC: 388.69, Prec. : 64.64%, Recall: 70.90%).

Attribute	Coefficient	Std. Error	Signif. code	Effect size
Branch	-0.005857	0.004467		-1.71518E-05
PersisVarietyLow	2.235964	0.468840	***	1.412526089
ActMaint	1.473443	0.374606	***	1.021655129
JDKStandard	-1.896858	0.449827	***	-0.78785475
QualDocMed	-1.599531	0.514350	**	-0.721445889
HierStruct	-0.784968	0.322907	*	-0.438725736

CHAPTER 6 ARTICLE 3: RUN-TIME CONFIGURATION-AS-CODE

Mohammed Sayagh, Noureddine Kerzazi, Fabio Petrillo, Khalil Bennani, and Bram Adams
Submitted to Empirical Software Engineering (EMSE)

Abstract: Maintaining software configuration options is a challenging task from the developers' point of view. While medium-to-large projects typically have a dedicated configuration engineering process consisting of a subset of 9 major activities, prior work has identified 22 challenges associated with these activities. While existing work has focused on individual challenges such debugging of configuration failures or determining the optimal default value of options, we instead discuss and empirical evaluate 4 basic principles for configuration engineering that show potential for addressing a large subset of the challenges. A large-scale user study with 55 participants and 11 tasks shows that the principles indeed significantly improve correctness and speed for 8 out of the 11 tasks, while obtaining a draw for the other tasks. These results are promising, especially since the improvements are independent from developer experience.

6.1 Introduction

Successful software applications and middleware are portable across platforms and adaptible to different environments and usage scenarios [172]. A major component shared by such applications is an elaborate software configuration system. Basically, each decision that depends on the user's platform, environment or usage scenario is postponed until actual deployment or even execution of the system (without recompiling), simply by replacing hardcoded numbers or string literals in the source code by so-called configuration options. Such decisions could include the selection of features that should be enabled, the definition of host names, or calibration of tuning parameters. Each option consists of a name, a type, default value, documentation and constraint (i.e., set of allowed values) determined by developers, and a final value determined by the end user (or for example DevOps administrator). An API allows developers to assign the value of an option to a variable, then to use this variable across the code instead of hardcoded values.

Real-life configuration systems are huge! For example, Sayagh et al. [163] report that Apache Hadoop 2.7.1 has 800 configuration options, while Mozilla Firefox 43.0 has more than 2,000! This scale brings substantial challenges because a configuration system does not come for free: configuration options at all times need to remain synchronized with the source code [125]

to avoid unconfigurable features or configuration choices without effect. To manage this traceability, software projects implement a configuration engineering process [121], consisting of 9 configuration-related activities, from the creation of a configuration option, choice of storage medium (e.g., configuration file or database) and option data type, to managing access of options in the code, comprehension and internal knowledge sharing of options, maintenance, debugging of configuration errors and quality assurance.

While configuration activities like debugging of configuration errors [185, 212, 215], choice of default values (part of option creation) [103, 219] and configuration-aware testing (part of quality assurance) [50, 95, 157] have been studied in depth, the other activities are not. In particular, in an earlier study [121] (currently under review) we identified 22 challenges (Table 6.1) related to the 9 configuration activities, based on 14 interviews and 229 survey responses. A systematic literature survey showed how most of these challenges are not yet addressed in practice.

Various interviewees and surveyees suspected that managing traceability between configuration options and the source code is the key to dealing with many of the challenges. Surprisingly, several of them formulated the same, simple principles that center around the concept of "run-time configuration as code". In other words, the definition of a configuration option (e.g., its name and type) should be integrated inside the source code instead of being physically separated, then configuration files for end users could be automatically generated, always in sync with the option definitions in the code. Similar to Infrastructure-as-Code [88], these principles allow known best practices, tool support and analyses for source code to be applied to configuration, such as automated testing, code review, refactoring support and smell detection.

This paper aims to present the 4 principles, then to empirically evaluate their impact on a user study involving 8 typical configuration engineering activities, evaluated across 11 user study tasks and 55 participants spread across industry and academia on two continents. Our main contributions are:

- Presentation of 4 principles of configuration engineering aimed at enforcing consistency between configuration options and source code.
- Large user study with 55 participants (industry/academia) and empirical analysis of the impact of the 4 principles on 11 typical configuration tasks.
- A prototype implementation of a configuration framework ("Config2Code") that implements the 4 principles.
- 41 screencasts (of up to 4h) of participants solving real configuration engineering tasks [161].

6.2 Background and Related Work

6.2.1 Software Configuration

Software configuration allows to postpone a decision in a software system until the required information is available, typically at installation or run-time [172]. These decisions can be made explicit in the user requirements [145], emerge during development to anticipate use cases of advanced users, be related to testing (enabling/disabling features on the fly), or pop up during the evolution of the code base. For example, agile developers often focus on getting the logic right for a specific case without being disturbed about generalization ("You Ain't Gonna Need It" principle). Later on, during refactoring, the functionality can be generalized by "externalizing" hard coded numbers and string literals into variables ("configuration options"), whose value is loaded from some kind of configuration storage medium, typically a file, database or the application's program arguments. Options can range from host names to tuning parameters, debugging options and feature toggles [88].

Figure 6.1 shows a .ini configuration file that assigns value `root` to configuration option *database.connection.username*. This file (storage medium) is managed by the end user. For this option (and its value) to take effect, an application typically needs to access the storage medium within a certain location of the code base (`ProviderClass`), then distribute the option's value to any other location in the code depending on it (`ConsumerClassX`). Dedicated libraries exist to help developers implement `ProviderClass`es and to provide APIs for accessing options within `ConsumerClass`es, but 46% of projects still roll their own coding support [163].

6.2.2 Related Work

The concept of Run-time Configuration-as-Code (RCaC) discussed and evaluated in this paper is tightly related to the popular concept of Infrastructure-as-Code (IaC) [88]. Both infrastructure and software development teams build and manage infrastructure using au-

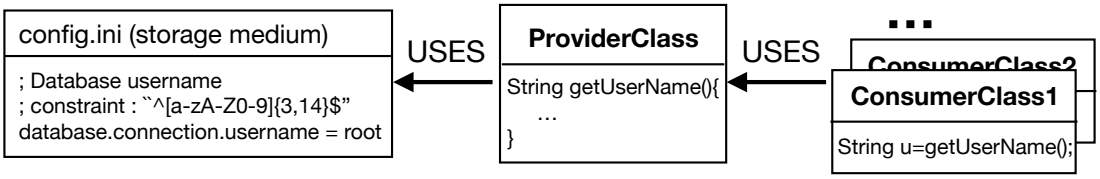


Figure 6.1 Example configuration system with one option.

tomated "Infrastructure as Code" (IaC) tools [122]. The infrastructure team is responsible for, amongst others, automatically building the environment[1] in which an application should be deployed [88]. In contrast, the development team traditionally has been responsible for developing highly-configurable software systems [172]. The research and practices related to both teams, while evolving separately for a long time, currently are converging under the influence of DevOps [48]. This is an extension of agile development that focuses on bringing developers and operators together in order to synchronize development and production.

This synergy between code and infrastructure has led to massive adoption by modern software organizations of IaC, in the form of declarative specifications in a domain-specific language like Puppet [108] and Ansible [40]. These specifications effectively are a form of source code, whose "compilation" in this case generates the desired environment (e.g., virtual machine). Researchers have indeed identified several source code phenomena in IaC code. Jiang et al. [92] studied the co-evolution of Puppet and Chef configuration files with source code, tests, and builds, and found a tight coupling of IaC file changes with test files. Sharma et al. [166] empirically studied a catalog of 13 implementation and 11 design configuration smells. The design configuration smells were shown to have 9% higher average co-occurrence among themselves than the implementation configuration smells, suggesting that the developers should pay more attention to the former. In contrast to the work on IaC, we focus on the synergy between the *run-time* configuration options of a software system and its code base.

Another line of related work primarily focuses on dealing with run-time configuration options and their related problems. We provide a systematic literature survey elsewhere [121], and instead focus here on the most closely related work. Zhang et al. [219] addressed the concern of finding the right option to be changed in order to obtain the software's desired behavior. They introduced a technique based on dynamic profiling and static analysis, supported by a tool called *ConfSuggester* to help debug configuration errors. Related to this, Huang et al. proposed *ConfValley* [85], a declarative language to express and check configuration specifications. Li et al. presented a tool called *ConfTest* [110] to prevent misconfiguration, and evaluated it against injected misconfigurations. Dong et al. proposed an approach called *ORPLocator* [67] to support detection of inconsistencies between source code and documentation via static analysis. They identify for each configuration option the source code locations reading it, then compare the results against the option names listed in the documentation. Similarly, Jin et al. presented *PrefFinder* [94], an NLP framework that navigates scarce, distributed documentation to find inconsistencies.

[1] An environment is the server, virtual machine or for example container on which an application will be deployed, providing the required operating system, 3rd party libraries and network connections [48].

Our study is fundamentally different from the above work on software configuration. Instead of focusing specifically on debugging of configuration errors or finding the best default value of an option, we study 4 principles of configuration engineering able to address a wide range of configuration challenges outlined in Table 6.1. The next section discusses these challenges in detail.

6.3 What is Problematic with Run-time Software Configuration?

While run-time configuration has been an ongoing concern in software development for decades [172], practitioners still suffer from a wide range of challenges involved with it. In particular, in order to provide and maintain a typical configuration system as shown in Figure 6.1, an organization needs to implement a configuration engineering process, i.e., a "discipline of activities involved in the integration and maintenance of configuration options in a software application" [121, 148, 185, 189]. For example, when a certain functionality should become configurable, one needs to add a new configuration option, pick a good name for it, then access the option's value within the code to decide when the configurable code should be enabled, or to tweak the code's behaviour in some other way.

In earlier work [121], we performed interviews with 14 experts, a large survey with 229 software engineers and a systematic literature review to recover and understand the typical configuration engineering process followed in practice, as well as to identify challenges and potential solutions. We distilled 9 major configuration activities and 22 related challenges, as summarized in Table 6.1. For each challenge, the table also indicates whether it is related to (M)anagement choices, (I)nherent difficulties of software configuration or (T)echnical details (e.g., addressable with tool support or implementation guidelines).

The (M)anagement-related challenges require explicit planning of options (e.g., based on requirements), clear assignment of an "owner" to each options to coordinate changes, evaluation and adoption of the right framework (library) to use to manage and access configuration options in the application at hand, a strategy to avoid configuration failures to regress (e.g., by documenting incorrect option values in a wiki), enforced guidelines for documenting options for end users and clear communication between developers regarding the goal and impact of options. Most of these challenges require process-level changes and follow-up, while some (such as the lack of option documentation [213]) could also benefit from better technical support.

In contrast, the challenges (I)nherent to software configuration cannot be avoided through better organization or tooling. For example, any added option enlarges the list of options

Table 6.1 Overview of challenges related to configuration activities [121], which are either (M)anagement, (I)nherent or (T)echnical. Here, we focus on the challenges in bold.

activity	challenge
1. creation of options	ad hoc planning of options (M)
	adding options increases complexity (I)
	choosing widely applicable default value (I)
	unclear configuration ownership (M)
2. managing storage media	**mixing media increases complexity** (T)
	choice of media impacts performance (T)
3. managing option type	**choice of option type** (T)
	configuration variants across environments (T)
4. configuration access in code	**coupling between ProviderClass and ConsumerClass** (T)
	adoption of dedicated frameworks (M)
5. comprehension of options	**unknown impact of option (change)** (T)
	lack of option comprehension tools (T)
	meaningless option names (T)
6. maintenance of options	**option removal is risky** (T)
	traceability between options and code (T)
7. resolving configuration failures	**debugging config. failures is hard** (T)
	lack of configuration debugging tools (T)
	no strategy for avoiding config. regression (M)
8. knowledge sharing	**lack of option documentation** (M)
	no internal communication about options (M)
9. quality assurance	**code review ignores configuration** (I)
	lack of automatic config. validation (T)

to read and understand [219], possibly frightening or at least puzzling potential users [212]. Similarly, the default value of an option has to be chosen as to enable plug-and-play functionality for most of the end users, which is surprisingly hard to achieve. The unclear link between configuration options and the source code impacted by it [147], as well as the focus of code review on changed lines only, make code review of configuration-related changes a challenge.

Finally, the (T)echnical challenges refer to more implementation-related challenges. They require more than just the selection and adoption of a dedicated configuration framework (which in itself is a challenge as well [163]). For example, the impact of an option on different parts of the code base [50], the need to choose meaningful option names (enabling easy understanding) and lack of automatic validation of constraints on the value of configuration options need substantial technical support [95].

Instead of focusing on one particular challenge, this paper explores and empirically evaluates four basic principles inspired by our earlier interviews and questionnaire [121] that have the potential of resolving most of the technical challenges as well as some of the management-related ones. The next section presents these principles, after which we report on a user study in which these principles are evaluated.

6.4 Run-time Configuration-as-Code

The major insight behind the 4 configuration engineering principles studied in this paper can be summed up by a quote from one of our industrial interviewees [121]: "*Configuration is Code Too*". As explained in subsection 6.2.2, this is certainly not a new principle as it also forms the backbone of Infrastructure-as-Code (IaC) [88]. While our interviewees were implying something similar for the run-time configuration (instead of the environment) of an application, according to them the solution used for IaC, i.e., a domain-specific language separate from the source code, would not suffice.

Indeed, while the environment of an application of course needs to be consistent with the source code of an application (e.g., by providing the specific API version of a third party library used by the code), this is even more so for the run-time configuration of said application. This run-time configuration needs to be able to turn on/off fine-grained features implemented by the source code, to initialize those features and even to define the values of constants assigned to variables in the code. This tighter coupling between run-time configuration and code (compared to environment and code) requires more attention to traceability and consistency between the run-time configuration and source code. Physically separating both is bound to lead to inconsistency issues, hence the term "run-time configuration-as-code" really implies *physically attaching* configuration options to the code they apply to.

We initially prototyped this idea and then presented it to experts during the next interviews in order to to get further feedback. We additional improved our prototype by incorporating more features after each of these interviews to end up with a set of 4 basic principles of configuration engineering that theoretically enable "run-time configuration-as-code" and as such have the potential to address a wide range of challenges in Table 6.1 (the bold ones). For each such challenge, Table 6.2 indicates which of the principles it is being impacted by, and we explain the conceptual reasons for this impact. In later sections, we relate on a user study in which these principles are empirically evaluated regarding their ability to address the challenges.

Table 6.2 Mapping the principles to the bold challenges of Table 6.1.

	Task	Configuration-as-Code	Encapsulation of Configuration Access	Generation of Configuration Media	Automatic Validation
clear configuration ownership		+			
mixing media increases complexity				+	
choice of option type	T1	+			
coupling between ProviderClass and ConsumerClass	T1 & T4		+		
unknown impact of option (change)	T4	+	+		
lack of option comprehension tools	T5	+	+	+	+
meaningless option names	T7				+
option removal is risky	T4	+	+	+	
traceability between options and code	T1, T2, T3 & T4	+		+	
debugging configuration failures is hard	T6	+			
lack of configuration debugging tools	T6	+			
lack of option documentation	T1			+	+
code review ignores configuration	T8	+	+		
lack of automatic configuration validation	T7				+

6.4.1 Configuration-as-Code

The first principle physically moves the definition of a configuration option and its related metadata (type, default value, description and constraints) from a user-space storage medium such as a configuration file to the developer-space ProviderClass (see Figure 6.1). This allows developers to remain inside the source code to create a new configuration option. Furthermore, by adopting a specific syntax or idiom to specify options, it is straightforward for tools or manual search queries to identify all options of an application or to serve as starting point for refactoring [56].

Figure 6.2 illustrates this on a Java application, using the syntax of our *Config2Code* prototype of the 4 principles. Here, a developer decided to make the database user name configurable by end-users by annotating the class attribute "username". The type of the option is determined by the type of the attribute (String), while the annotation "*@Config*" specifies a namespace for the option, a default value, constraint and the desired type of storage medium ("*support*").

Apart from reducing the need for context switches and improving traceability, this first principle also has other benefits. For example, ownership of an option is now determined through code ownership of the code file it is defined in. Furthermore, determining the impact of an option is now possible by using regular code analyses and tools used to determine the impact of a variable (forward direction), to determine the (configuration) variables impacting the code location of a configuration failure [143] or to safely remove a configuration option from the code without causing undesirable code paths to become active. Finally, given that the definition of configuration options is now part of the code, any changes to such a definition are now captured in the version control system as regular code commits. Since these are the

```
public class Hello {
        @Config(name = ``username'',
                namespace = ``database.connection'',
                comment = ``Database username'',
                defaultValue = ``root'',
                constraint = ``^[a-zA-Z0-9]{3,14}$'',
                support = Config.FILE // Config.ARGS or Config.SYS
                )
        private String username;
}
```

Figure 6.2 Illustration of principle 1 (Configuration-as-Code), using the syntax of *Config2Code*.

commits considered by code review, run-time configuration changes can become an integral part of the review process instead of a special case.

6.4.2 Encapsulation of Configuration Access

Bringing the definition of configuration options inside the ProviderClass is only one step. The second principle focuses on the manner in which the values of these options can be accessed by the rest of the code, in particular the ConsumerClasses. Based on common software engineering sense [172], it is no surprise that the principle requests a well-encapsulated API for accessing configuration values, reducing coupling and duplication within the application. Note that it does not suffice to just adopt a third-party configuration framework like *jConfig* or *Preferences*, since scattered usage of such a framework's API throughout an application leads to strong coupling and complicates later migration to another framework.

Apart from reducing coupling between configuration and the application, encapsulated access results in a uniform API throughout the application, again making it easier to determine the impact of a given option and whether an option can be safely removed. While principle 1 has made the definition of configuration options explicit in code changes, the usage of these options across the code base now is also made explicit through an API, further encouraging systematic code review of configuration option-related changes.

6.4.3 Generation of Configuration Media

While principle 1 brought the definition of configuration options into the code and principle 2 distributes the value of these options in a disciplined way, principle 3 closes the loop by automatically generating an appropriate storage medium for the end user (e.g., a .ini file in Figure 6.1). This medium contains the current set of configuration options and their metadata (including a default value) and is always synchronized with the options that are currently used (defined) in the code by the developers, both in terms of option name, type, documentation and constraints. This generation also allows end users (or operators) to easily compare the previous version of the medium (containing the user's custom values assigned to options) to the new version in order to detect new options, removed options, changed constraints or option types, etc.

This principle also allows the type of storage medium to be easily changed, and different types of media could be mixed for different subsets of the options (a configuration file for some options, while command line arguments for others). Furthermore, the principle also achieves full and automated traceability between options and code, and helps to address the challenge of missing option documentation, especially when combined with principle 4.

6.4.4 Automatic Validation

Principle 4 requires automatic validation of the values assigned to configuration options as well as of the options' definitions:

- Managers or technical leads can specify norms for configuration options that should be respected by developers, for example a specific naming convention.
- Each option should only accept values of a certain type, for example an IP address (4 numbers from 0 to 255) vs. a hostname (textual string).
- Options could be subject to specific business rules that cannot simply be expressed in terms of a variable type, for example "the random number seed should have < 3 digits".

Furthermore, this validation should be performed automatically, either during a build of a new version of the source code (developers) or during program start-up (end users). If a violation of a constraint is detected, the system should either halt or fall back to option values that are known to be good.

As an example, Figure 6.3 shows a static checkstyle rule for option names specified by the team lead of the example in Figure 6.2. The rule encodes that the name of an option is mandatory and should consist of 3 to 12 letters (upper- or lowercase), and it specifies an error message (not just a warning) in case of a violation. Furthermore, during compilation

```
<configChecker>
  <module name="name">
   <property name="mandatory" value="1" />
   <property name="format" value="^[A-Za-z]{3,12}$" />
   <property name="type" value="error" />
   <property name="message" value="Incorrect name format" />
  </module>
</configChecker>
```

Figure 6.3 Illustration of principle 4 (Automatic Validation), showing checkstyle rules encoding programming conventions for configuration options, for the example in Figure 6.2.

the type of the option will be enforced (since it corresponds to the type of a class attribute). At run-time, when the value assigned to an option is read from the storage medium , the constraint specified on line 6 of Figure 6.2 furthermore will check that the password will consist of 3 to 14 alphanumeric characters.

Automatic constraint validation improves comprehension, and enforces naming conventions and the presence of documentation.

6.5 Design of User Study

The purpose of this paper is to empirically evaluate how well the 4 configuration engineering principles of the previous section are able to support developers with typical configuration engineering tasks.To this end, we carried out a controlled experiment [205]. Controlled experiments have been widely used in human-computer interaction to perform user evaluation, and more recently in software engineering as well [62, 202]. The design of our user study carefully follows the 5-step methodology described in [119].

6.5.1 Research Questions

Our controlled experiment compares a configuration engineering framework implementing the 4 principles to a base framework in order to address the following two research questions across a wide range of configuration engineering tasks:

- RQ1: Do the principles increase the correctness of configuration engineering tasks?
- RQ2: Do the principles reduce the time needed to complete configuration tasks?

We analyze the results of these questions both quantitatively and qualitatively, and also consider the impact of participant experience (confounding factor). The remainder of this

section first presents our *Config2Code* prototype of the 4 principles, and the study object, then discusses the design of the experimental tasks, followed by the choice and composition of subject groups, and finally our experimental protocol.

6.5.2 Config2Code

To evaluate our approach, we built a Java framework called *Config2Code* that implements the 4 principles. We have already seen its syntax in Figure 6.2, and it comes with a Java builder plugin that can be used within Eclipse within Maven builds. The plugin basically parses the "*@config*" annotations, then uses Javassist to inject the value of configuration options into the annotated class attribute at the bytecode level. Furthermore, the annotation metadata is used to automatically generate the right configuration storage medium.

For automatic validation, the current version of *Config2Code* only allows for regular expression-based constraints that are expressed in terms of characters. Both the regular expression constraints and manually written checkstyle rules are used to automatically validate constraints on the values of options. Integrating more powerful Z3 constraints is future work.

6.5.3 Study Object

As study object, we looked for an open-source GUI application with a non-trivial number of configuration options and that is not following the 4 principles to manage its configuration engineering concerns. We focused on a GUI application such that configuration changes would easily be visible to study participants. Furthermore, the source code of the application should be large enough to be challenging, but should be structured well enough so as not to overly divert the participants' attention from the configuration.

Eventually, we selected JabRef [21], which is an open source tool dedicated to managing BibTeX references. Two of the authors are familiar with this software project from a previous study. Version 4.0 of the application consists of 171,233 lines of code, 1,421 classes and 177 options. Hence, considering the size of the application, the number of configuration options is substantial. By default, JabRef uses the *Preferences* configuration framework. This is a basic framework that comes bundled with the Java SDK since Java 4. It reads configuration options from a configuration file, and allow developers to use options via an API, whose methods allow to read different types of configuration options (String, int, double, ...). These methods take as argument a configuration name and a default value, which is returned in case the accessed option is not declared in the configuration file. Preferences is one of the most popular existing Java configuration frameworks [163].

The *Preferences* framework as used by JabRef was not implementing the 4 principles:

1. Options are specified in textual configuration files.
2. JabRef is strongly coupled to *Preferences*, as it calls Preferences methods API (getInt, getString, ...) throughout the code with the requested option as String argument.
3. The textual configuration files are maintained manually.
4. No validation is performed of option values.

In order to use JabRef for our study, we made two important changes. First, we removed the configuration GUI from the menu and code base in order to allow subjects to focus only on the source code and the external (*Preferences*) configuration file. Second, we prepared a *Config2Code* version of JabRef by entirely removing the use of *Preferences* and replacing it with *Config2Code* (i.e., all the 177 *Preference* options were replaced by *Config2Code*).

6.5.4 Task Design

The design of our experimental *Tasks* was driven by the 14 bold challenges of Table 6.1. Hence, our initial goal was to formulate one study task per challenge, yet some challenges are related and we also wanted to reduce the typical time to finish all tasks to a reasonable number of about 1h30. This is why we ended up with 11 tasks spread across 8 configuration engineering activities. Table 6.3 provides descriptions of the 11 resulting tasks. The handouts we gave to participants are available online [10, 11].

6.5.5 Participants

We initially designed our experiment to consider four categories of subjects, across two dimensions: {Industry, Academic} and {Novice, Expert} (Table 6.4). Novice industrial participants are subjects working in industry with less than 3 year of experience, while industry experts have more than 3 years of experience. Novice academic participants are undergrad engineering students, whereas expert academic participants are students pursuing a master or Ph.D.

One of the main barriers for controlled experiments of software engineering tools is participant recruitment [54], especially due to software professionals being busy [106]. To complement the 2 industry participants that we contacted based on personal contacts, we also contacted 5 remote freelance developers on Freelancer.com. We controlled for several well-known issues involving remote participants [106]. First, since developers tend to inflate their level of experience, we interviewed all candidates via chat and we asked them to perform a warm-up task via which we measured and evaluated all skills requested from the developers. Only

participants who correctly completed the warm-up participated in the experiment. Second, to deal with developers who may temporarily suspend a task or would not follow the instructions correctly, we asked all participants to record a video screen cast during their working session. We excluded participants that did not record the video. Third, high payments could attract participants that are excessively motivated by money, which may lead to unrealistic behaviour [106]. To mitigate this, we paid Freelancer.com's median flat sum of CAD$35 per experiment (in line with Ko et al.'s US$30 [106]) after the work was completed and we were 100% satisfied with its quality.

All of the academic participants were volunteers, which we invited to participate in exchange for a bonus in their courses and a certificate acknowledging their participation. The students were recruited at one North American and one North African university during the Summer of 2017 and Winter of 2018. None of them had prior experience with *Preferences* or *Config2Code*. In addition, we also had 7 student and one expert participants who participated in pilot runs of the study in order to refine the questions and study protocol. They are not included in Table 6.4, nor in the results section.

While our study design targeted the impact of both the {Industry, Academic} and {Novice, Expert} factors, we eventually dropped the former. First of all, we had an unbalance between industry and academic participants, with a ratio of 13 to 42. While this problem was not unsurmountable, we noticed that many students, even novices, had prior experience developing software. In some cases, students had more experience than (novice) freelancers. In one case, an academic expert (PhD student) had only 1.5 years of Java experience compared to an academic novice with 5 years of Java experience. In order not to derive wrong conclusions, we elided the {Industry,Academic} factor and only consider {Novice,Expert} in our discussion. This is why we eventually only consider the number of years of Java experience as a metric for experience, i.e., a participant with 3 or more years of Java experience is considered as an expert, whereas participants with less than 3 years are novices. This decomposition is shown in the "All" column of Table 6.4.

6.5.6 Experimental Protocol

Participants were randomly assigned to either the experimental group (using *Config2Code*) or the control group (using *Preferences*). We used stratified sampling based on the {Industrial , Academic} and {Novice,Expert} factors, resulting in the composition of Table 6.4.

The experiment was performed using two types of virtual machines (VMs): *(1)* VMs hosted on Google Cloud for non-local participants, and *(2)* a similar environment on VirtualBox installed on our lab machines for local subjects. Each VM was set up with Eclipse and either

Config2Code or *Preferences*. To analyze the subjects' results, the screen of the VMs were recorded via the Cattura Google plugin or by configuring Virtual Box to capture the screen of VMs.

Before performing the 11 tasks, both the experimental and control groups received an introduction about configuration options in general, the specific framework they were going to use, and the three steps of the experiment (warm-up, experiment, and exit survey). We did not divulge our intent to compare *Preferences* and *Config2Code*. The participants then started with a warm-up exercise on a toy project, in which they could learn that taught them how to use the framework they were assigned to. The results of this exercise were just used to filter out participants that were unfit for the study. These are not included in the numbers of Table 6.4.

Once finished with the warm-up exercise, the participants would enable screen recording of the VM and start the 11 tasks. We warned them that it might typically take 1h30 to finish all tasks, yet they were free to stop at any time. All sessions of experiments were supervised by one of the authors to enable clarification questions, if needed. 6 of the 11 tasks required a written answer in a separate response file saved in the desktop of each virtual machine. Once finished (or when quitting), the screen recording would be stopped, and the participants had to fill out an exit survey in which they express their impression about the experiment in general, and about the advantages and challenges they faced during the experiment and that are related to the framework they used.

Finally, in order to address the research questions, we marked the modified source code and the answers in the response files of the 11 tasks to determine correctness. We defined a check-list to mark each task. For example, for T1.1 and T1.2, participants should (1) define the new options, (2) comment them, (3) define their constraints, and (4) use them within the source code.

In order to determine the time needed to perform the individual tasks, we scrolled through each participant's video to record the moments on which they switched to the next task. A task starts when a subject finishes reading its requirements and finishes when she completed the code or answered the exercise on the responses file. The challenges we encountered were the length of the videos (up to 4 hours) as well as the fact that some participants answered some questions in more than one shot. They would start a given exercise and come back to finish it later on or even at the end of the experiment, hence we had to analyze the entire video to be sure.

6.6 Quantitative Results

This section discusses our empirical evaluation of the impact of the 4 principles on the ability of developers to perform typical configuration engineering tasks. The next section then discusses those results for each individual task.

RQ1: Do the principles increase the correctness of configuration engineering tasks?

Motivation: The 4 principles discussed in section 6.4 are conjectured to enable developers to perform configuration engineering tasks more correctly compared to not following them. Therefore, we define our null hypothesis as:

H0: There is no significant difference in task correctness between Config2Code and Preferences participants.

Approach: Based on the textual answers to each task and the modified source code of each participant [161], we assigned, for each task, a mark to the participant. Since subjects were free to leave at any point or skip any question, we ignore the tasks that they did not perform (instead of giving a zero score for those). For this reason, we analyze the marks of each individual task rather than calculate a global score, and we count the number of wins, losses and draws for *Config2Code* compared to *Preferences*. Similary to Wettel et al. [201], wins, losses and draws were determined using the (non-parametric) Mann-Whitney-Wilcoxon test, with a confidence level of $\alpha = 0.05$.

In a second step, we evaluated the impact of participants' experience on the results (i.e., experts vs novices). For this, we compared the results of *Config2Code* experts against *Config2Code* novices on the one hand, and *Preferences* experts against *Preferences* novices on the other hand. This comparison again is at task-level and based on Mann-Whitney-Wilcoxon tests with $\alpha = 0.05$.

Results: *Config2Code* wins in 8 tasks (T1.1, T1.2, T2, T3, T4, T5.1, T5.2, and T7.1), and draws in the other 3 tasks (T6, T7.2 and T8). Table 6.5 indeed shows how we can reject the null hypothesis ($p < 0.05$) for 8 out of the 11 tasks, while we were unable to for the other tasks. As shown by the median scores, there was no task for which the *Preferences* subjects performed more correctly than *Config2Code* subjects. Indeed, *Config2Code* increased median correctness with up to 300% compared to the corresponding *Preferences* scores for task (T5.2), in which participants had to find possible option values online. For T8, more than half of *Preferences* were unable to perform the task correctly, leading to a median score of 0, compared to 1 for *Config2Code*.

As highlighted in Table 6.5, we found that overall the results are not impacted by the experience level of participants ($p \geq 0.05$), except in the case of T1.2 and T5.2 for *Preferences*. The difference between T1.1 and T1.2 is to check the correctness of the new option via a regular expression constraint, which was not easy for novice developers. T5.2 was also not easy for novice developers, which were not able to find possible values of a configuration option. From our observation to the task T1.2, we found that *Preferences* participants had hard time to identify where they should check the constraint within a large project size, and how to check a regular expression constraint in Java. This is why many *Preferences* novice participants had to check online how to use a regular expression. Similarly, for T5.2 novice participants did not have any concrete strategy to find possible values of a configuration option, they rely just on searching for that option's key on internet.

RQ2: Do the principles reduce the time needed to complete configuration tasks?

Motivation: The goal of this research question is to evaluate if *Config2Code* helps developers performing configuration tasks faster than *Preferences*. Therefore, we define our null hypothesis as:

H0: There is no significant difference in time required between Config2Code and Preferences participants.

Approach: For each task, we measured the time required by each participant, filtering out participants who did not successfully complete a given task. We defined "successful" completion as obtaining at least half of the marks for that task, i.e., a score ≥ 0.5. Note that this also led to filtering out participants who forgot to start or prematurely ended screencast recording, since no timing information was available for them. Finally, we also analyzed the impact of subjects' experience on the required time to solve a task.

We observed in the videos that participants solve both tasks T1.1 and T1.2 in parallel. They add both configuration options either in the configuration file for *Preferences* or as two annotations for *Config2Code*, then change the source code to use both configuration options. Therefore, we report a single time measurement for both.

Results: *Config2Code* wins on 6 tasks (T1.1, T1.2, T2, T5.1, T5.2 and T7.1), draws for the other 4 tasks (T3, T6, T7.2 and T8), and loose in only one task (T4). As shown in Table 6.5, we again did not find any cases for which *Preferences* developers performed significantly faster than *Config2Code*. In fact, *Config2Code* helps to save up to 94.62% (in task T5.2) of development time compared to Preferences. Task T4, however, does

lead to a low p-value of 0.02, with a median slowdown of 90% compared to *Preferences*, even though the results for *Config2Code* in RQ1 were more correct for this task.

In the other cases, the results for RQ1 and RQ2 match each other. For example, *Config2Code* developers can immediately understand a configuration option, with a reduction of 94.62% and 79.67% of time (T5.2 and T5.1). *Config2Code* developers are also able to save 20.18% of time to change an option name (T3), while for T1 they require a median of 1,559.5 seconds (25.99 minutes) to add an option with *Config2Code* instead of 3,034 seconds (50.95 minutes) . Finally, the RQ2 results are not impacted at all by the experience of developers.

6.7 Qualitative Discussion

This section qualitatively discusses RQ1 and RQ2 for each task.

T1: Creation of Configuration Options: As shown in Table 6.6, 73.07% of *Preferences* participants did not add configuration constraints that check the correctness of the value of an option, compared to only 3.44% of *Config2Code* subjects. In addition, 53.84% of *Preferences* subjects forgot to comment their configuration options. Because JabRef defines a map data structure of default option values within its source code, the *Preferences* participants had to define the default values not only in the configuration file, but also in that defaults map. This is why 7.69% of *Preferences* participants forgot to define default values in both places.

Therefore, **principle 1** (i.e., putting all option-related information inside the source code in one location) has had a positive impact on the correctness of adding a configuration option as well as on the time required (see Table 6.2). On the other hand, using a configuration option requires writing additional code in each ConsumerClass for both *Config2Code* as well as *Preferences* participants, which is why similar percentages of participants (30% and 33%, respectively) forgot usage sites of the new configuration options.

From the videos, we observed that both groups of developers liberally used copy-paste of existing option definitions. While *Config2Code* participants could just copy and modify annotations in one source code area (that they first had to find), while the *Preferences* subjects had to perform many more steps across different code areas. We observed that they initially created the options in the configuration file, added the new option names as two global constants, then added the default values to the right map, in order to then use the options in the right files. Given the complexity of all these steps, we noticed substantial trial-and-error for this group. One *Preferences* participant confirmed: *P35: "Seems to have a lot of needless steps between getting a config value and using it"*, while one *Config2Code* participant found: *P33: "It's easy to add a config variable and assign it in the config.ini file"*.

The difference in percentage of developers adding constraints, together with the large number of erroneous options added by the *Preferences* group, show how **principle 4** helps developers to ensure correctness of new options. The exit survey confirmed: *P33: "It's also easy to give extra constraints on the different fields of config variables once you have a working example".* Finally, principle 3 helped participants by automatically regenerating configuration files after each modification.

T2: Changing Default Value: Changing the default value with *Config2Code* requires only changing the attribute "defaultValue" within the ProviderClass for *Config2Code* subjects, while it requires changing the default map within the source code and the configuration file for the *Preferences* subjects. Forgetting to change the default value in both places is error-prone, since JabRef seemingly would be using a different value than the one specified in the configuration file (i.e., the one listed in the map data structure). Due to the context switches between the configuration file and the source code, 90% (19/21) of *Preferences* developers modified just the configuration file or just the defaults map, while this task was straightforward for 76% (20/26) of *Config2Code* participants due to **principle 1**.

T3: Changing a Configuration Name: JabRef stores its option names as string literals in global constants within the class *JabRefPreferences.java*, which is by its turn used to get access to the configuration file via the framework *Preferences*. Refactoring an option's name then requires developers to change that name within the class *JabRefPreferences.java* and also in the configuration file. Changing only one of these two artifacts can introduce an inconsistency between configuration files and the source code, and then *JabRefPreferences.java* can read an option that does not exist in the configuration file. Because *Preferences* does not help to automatically synchronize source code with configuration file (i.e, **principle 3**), *Preferences*' participants have to change both artifacts comparing to *Config2Code* subjects that have to change only the annotation Config. The reason for which we obtained a statistically significant difference in the correctness of task T3 between *Config2Code* and *Preferences*. Therefore, **principle 3** has a positive impact on refactoring option names.

T4: Removing Configuration Options: Participants had to replace 5 configuration options (O1...5) by another one (O6). An additional complexity (for both groups) was the fact that these options were accessed by reflection. The difference in correctness (and to some degree time) between both groups for this task was due to **principle 2**. While *Preferences* accesses these configuration options from the ConsumerClass without any encapsulation, *Config2Code* uses the ProviderClass accessors, either directly or via a reflective call from a ConsumerClass.

Therefore, due to the lack of such encapsulation, *Preferences* users faced the problem of having to search the code for all usage sites (Consumer classes) of these 5 options, as well as to remove them from both the configuration file, the default value map, and their names from the global constants (as discussed in T3). In contrast, *Config2Code* developers just had to modify the ProviderClass accessors to use O6 internally, and physically remove the annotation "*@Config*" to avoid synchronizing the source code with the configuration file. One *Preferences* participant said: *P37: "some code refactoring tasks need to analyze the code deeply".*

That said, we did observe in the videos that some *Config2Code* participants initially removed the "@Config" annotation, and the (previously configurable) class attributes and its accessors. However, this yielded exceptions due to the hidden reflective call. These *Config2Code* participants first tried to fix that unexpected bug, before realizing the simpler solution (modifying the ProviderClass accessors instead of removing). That explains the substantial slowdown compared to the *Preferences* group.

T5: Comprehension of Configuration Options: As shown in Table 6.5, there is a significant difference in correctness between *Config2Code* subjects (who were able to find exactly the possible values within the "*@Config*" constraint, i.e., **principle 1**) and the *Preferences* subjects, who had to search where that option is used within the source code or even to use online documentation. This especially was clear for T5.2. While some participants found some values online, their responses were inaccurate, as many values are simply ignored by JabRef. This would only be clear by exploring the source code, which is time-consuming. Due to the automatic synchronisation of option information (**principle 3**), *Config2Code* participants always found the up-to-date set of option values.

T6: Fixing a Configuration Error: This task saw draws for both correctness and time. The videos showed how participants would usually start out by guessing a likely incorrect candidate option (in terms of its value) related to the bug symptoms that they faced. In this case, because the bug was related to the auto-complete functionality, most of the subjects started by searching configuration options that have the keyword "autocomplete" in their names, then manually inspect their values. **Principle 1**, which is the only principle addressing the challenges "debugging configuration failures is hard" and "lack of configuration debugging tools" in Table 6.2, clearly does not suffice in this context.

A special characteristic of the configuration error addressed in T6 is that it was a functional error without any exception or error message, only showing a graphical effect. In future work, we plan to investigate configuration errors with an explicit error message as well as

performance-related configuration errors, both of which would allow source code inspection techniques to be used.

T7: Configuration Quality: This task showed a significant speedup and correctness for *Config2Code* (T7.1), but draws for both correctness and time of T7.2. An initial qualitative analysis of the videos showed that some *Preferences* subjects manually inspected each configuration option, whereas other participants copy and paste the whole configuration file in online regular expression checkers to solve this task. This is time-consuming compared to Config2Code, where developers could declare the regular expression within a checkstyle rule (cf. Figure 6.3; **principle 4**). Only in one *Preferences* case, a developer wrote a script to identify the number of options that do not respect the proposed naming convention.

Due to the large number of configuration options that do not respect the configuration naming convention that we proposed in this task, it was not difficult (in terms of correctness and time) for both groups to identify a set of examples in T7.2, explaining the draws for correctness and time.

T8: Configuration Review: Finally, we also did not observe any differences for the patch reviewing task. We believe that this is due to the limited size of the patch that we studied, making it an easier task for both groups. Furthermore, since the patch merely adds a new configuration option, all the information regarding the option's definition is included in the patch. A patch that would impact code that depends on an option (a fact probably not obvious from the commit's diff alone) likely would not change other data of the option and hence might be harder to review.

6.8 Threats to Validity

Threats to *internal validity* concern alternative factors that could have influenced our findings. One threat is the degree of competence of our subjects. To mitigate this threat, we ensured that participants are comfortable with Java having carried out at least one medium project with it. Second, we used randomization to fairly assign participants to treatment groups. For each group, we presented the purpose of the study and provided a warm-up exercise to be sure they have a good knowledge of the application domain. Furthermore, participants from both controlled and experimental groups could ask questions during the experiments. Third, the subjects may not have been correctly motivated. This thread was mitigated by the fact that all participated on a voluntary basis and received certain incentives (subsection 6.5.5).

We also recognize a threat related to the design of the experimental tasks, which may have been biased to the advantage of either of the two groups [205]. To alleviate this threat, we aligned the design of tasks with the configuration engineering tasks and challenges identified in earlier work [121]. Two of the authors were part of a pilot study that assessed the perceived task difficulty and time pressure, then to validate that we are measuring the right metrics.

Threats to *external validity* concern the generalization of our findings. Future work should consider other activities and tasks than those studied here, covering configuration engineering activities and challenges not covered by the current study. Furthermore, other object systems should be used, leveraging other standard configuration frameworks than *Preferences*. Finally, other participants should be recruited, involving more industry practitioners.

A second external threat to validity concerns the impact of other implementations factors instead of our four principles on our findings. To mitigate this risk, we conducted a qualitative analysis of the video records and discussed with our subjects after finishing their tasks.

Threats to *construct validity* consider agreement between a theoretical concept and a specific measuring procedure. One threat considers the experimenter effect: since we are both the authors and experimenters, this may have influenced any subjective aspect of the experiment. Although we are aware that we cannot exclude all the possible impacts of these threats, we did try to mitigate them by designing a checklist and a model of the answers with a grading scheme. Moreover, two of the authors performed the grading.

Finally, future iterations of this study should provide an automated way for participants to have the video started and stopped when starting or ending a task. This would avoid having to filter out participants without timing information.

6.9 Conclusion

This paper presents and empirically evaluates 4 configuration engineering principles via a user study with 55 participants and 11 tasks (spanning 8 configuration activities). Our findings show how the principles are able to improve correctness, speed, or both of them for 8 out of 11 tasks compared to a baseline configuration framework, while no statistically significant differences are observed for the other tasks. Especially for debugging tasks the principles did not show a real improvement. Furthermore, the improvements in correctness and speed were observed independent from participant experience, i.e., even novice developers benefited from the 4 principles.

While we prototyped the 4 principles in the *Config2Code* framework, they are in fact independent. For example, any software project could address principle 2 simply by designing an

explicit ProviderClass with a clear API, which would positively impact 5 of the configuration engineering challenges. Similarly, any framework using annotations might be a good candidate for implementing principle 1. On the other hand, principles 3 and 4 are less common in today's frameworks and our results suggest that it is worth investing effort in them.

Table 6.3 The 11 tasks administered in the user study.

Task	Description
T1. Creation of configuration options	Participants should create two new configuration options (one boolean **(T1.1)** and one integer **(T1.2)** option) based on requirements about the name of these options, their comments, default values, constraints for user choices, and the code area that should be modified to use the new created options.
T2. Refactoring - Changing a default value	Participants should change the default value of a configuration option. While this requires *Config2Code* users to change only an attribute of the *@Config* annotation, *Preferences* users need to change the code in two different places for this (the configuration file and a map data structure with default values).
T3. Refactoring - Changing option name	Participants need to change an unclear configuration name. While a simple change for *Config2Code* participants, *Preferences* participants need to change both the code where this option is used and the configuration file.
T4. Refactoring - Removing configuration options	Participants need to remove 5 similar options and replace them by one unique option. *Config2Code* participants can just remove the "*@Config*" annotation, and change the corresponding class attributes' accessors to instead use the proposed option. *Preferences* participants need to remove the option from the configuration file, then search the code to find where each option is used to replace it by the proposed option. This task is not trivial, as some of the option accesses use Java reflection.
T5. Comprehension of options	Participants had to identify the range of possible values of two configuration options. The first option **(T5.1)** is a JabRef specific configuration option that is related to the auto-completion of textual fields in the GUI, whose allowed values are specified inside the JabRef code base. The second option **(T5.2)** is related to the possible font styles that can be used. The latter option's values can be found online, yet not all of those are actually used by JabRef. Hence, participants should only report the used ones.
T6. Fixing a configuration error	Participants need to identify which option is responsible for a configuration error (incorrect value), then to fix the error (and prevent future errors of this option). This fix requires adding a "*@Config*" constraint for *Config2Code* or adding an if-check for *Preferences*.
T7. Configuration quality	The goal of this task is to check which options do not respect a predefined naming convention **(T7.1)** and to propose 5 examples of these options **(T7.2)**
T8. Configuration review	This task requires participants to review a patch of a newly created option, whose definition contains two problems: the option did not have any constraint attached (despite the commit message clearly mentioning the constraint), and the default value did not respect the constraint of the commit message.

Table 6.4 Decomposition of the user study subjects.

	Industrial		Academia		All		Total
	Novice	Expert	Novice	Expert	Novice	Expert	
C2C.	4	3	17	5	20	9	29
Pref.	3	3	16	4	18	8	26
Total	7	6	33	9	38	17	55

Table 6.5 Summary of the RQ1 correctness and RQ2 time results. NaN indicates that experts and novices of *Config2Code* obtained exactly the same results, while the median improvement values are relative to the *Preferences* results.

		T1.1	T1.2	T2	T3	T4	T5.1	T5.2	T6	T7.1	T7.2	T8
Correctness	Wilcoxon p-value	1.96e-05	8.53e-06	9.84e-07	0.037	3.95e-07	8.25e-06	6.99e-08	0.35	0.042	0.83	0.15
	Median *Config2Code* Score	1	1	1	1	1	1	1	1	0.75	1	1
	Median *Preferences* Score	0.625	0.75	0.5	1	0.5	0.5	0.25	1	0.5	1	0
	Median Improvement	+60 %	+33.33 %	+100 %	0 %	+100 %	+100 %	+300 %	0 %	+50 %	0 %	Inf %
	p-value Experience (*Config2Code*)	0.83	0.83	0.72	NaN	0.40	0.56	0.54	0.51	NaN	0.54	0.27
	p-value Experience (*Preferences*)	0.035	0.00059	0.18	0.13	0.29	0.10	0.025	1	0.56	0.35	0.64
	#*Config2Code* Participants	29	29	26	27	27	29	28	24	15	24	22
	#*Preferences* Participants	26	26	21	19	19	21	21	11	12	17	17
Time	Wilcoxon p-value	4.50e-07		0.0041	0.95	0.020	0.0014	0.0025	1	0.0061	0.23	0.44
	Median *Config2Code* Time (s)	1,559.5		40	267	408.5	37	24	312.5	171.5	57	564
	Median *Preferences* Time (s)	3,034		132	334.5	215	182	446	191	379	94	424
	Median Improvement	-48.60 %		-69.70 %	-20.18 %	+90 %	-79.67 %	-94.62 %	+63.61 %	-54.75 %	-39.36 %	+33.02 %
	p-value Experience (*Config2Code*)	0.21		0.89	0.17	0.35	0.86	0.097	0.15	0.34	0.49	0.89
	p-value Experience (*Preferences*)	0.66		0.68	0.64	0.90	0.69	1	1	1	0.32	0.4
	#*Config2Code* Participants	24		23	23	20	22	22	14	12	16	9
	#*Preferences* Participants	17		15	12	9	8	4	7	5	11	5

Table 6.6 Percentage of participants forgetting to add constraints, comments, usage sites and default values during the creation of configuration options (T1).

	%Constraint	%Comment	%Usage	%Default
Config2Code	3.44	3.44	27.58	0
Preferences	73.07	53.84	26.92	7.69

CHAPTER 7 ARTICLE 4: MULTI-LAYER SOFTWARE CONFIGURATION - EMPIRICAL STUDY ON WORDPRESS

Mohammed Sayagh and Bram Adams
Published in the 15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)

Abstract: Software can be adapted to different situations and platforms by changing its configuration. However, incorrect configurations can lead to configuration errors that are hard to resolve or understand, especially in the case of multi-layer architectures, where configuration options in each layer might contradict each other or be hard to trace to each other. Hence, this paper performs an empirical study on the occurrence of multi-layer configuration options across Wordpress (WP) plugins, WP, and the PHP engine. Our analyses show that WP and its plugins use on average 76 configuration options, a number that increases across time. We also find that each plugin uses on average 1.49% to 9.49% of all WP database options, and 1.38% to 15.18% of all WP configurable constants. 85.16% of all WP database options, 78.88% of all WP configurable constants, and 52 PHP configuration options are used by at least two plugins at the same time. Finally, we show how the latter options have a larger potential for questions and confusion amongst users.

7.1 Introduction

Configuration is the means to adapt a software application to different contexts and environments. For example, the Linux kernel can be customized to different users by selecting only the features that are of interest. Similarly, the kernel can be customized to a specific hardware platform by providing the details of processor, hard disk, and other devices. Each software system has its own mechanism for configuration, ranging from hardcoded constants to global variables, property files or dedicated databases, typically with a graphical user interface to hide the underlying storage mechanism.

An incorrect value of a configuration option could result in incorrect behavior of a system, which we refer to as configuration errors. Such errors occur often, typically are severe in nature, hard to debug, but they are actionable [219]. Such bugs are severe because they can have a catastrophic impact. For example, due to a misconfiguration, Facebook was left inaccessible for about two hours[1], depriving more than 500 million users from access to the

[1]https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919

Facebook website. Configuration errors are also hard to debug, since they need expertise in the failing application. However, if one is able to track down the cause of such an error, then the error is actionable since a maintainer just needs to update the configuration, usually without recompiling.

The major challenge for resolving software configuration errors is to find the violating configuration options, a challenge that is aggravated in multi-layer systems. Multi-layer systems consist of multiple layers, each of which hides the complexity of a lower layer, and has its own objects and configuration mechanisms. Since the behaviour of the system as a whole requires neighbouring layers to collaborate, one needs to understand each layer's configuration as well as how configuration options in each layer interfere with each other.

Let's consider the case of WP (Figure 8.1), which is currently the most popular content management system, and a typical example of a multi-layer system consisting of a LAMP stack (Linux, Apache, MySQL and PHP), the WP PHP application and a myriad of WP plugins. One example of a cross-layer configuration error was the inability of WP plugins to send emails, due to a misconfiguration in lower layers related to the PHP configuration option *sendmail_path*[2]. A second example was the case where the *NextGen* plugin was no longer able to upload images[3] until someone pointed out that the script downloading the images was blocked by a configuration option in the PHP layer (*memory_limit*). In both examples, configuration options in lower layers impacted the behaviour of the top layer plugins.

Whereas existing work focuses on software configuration and configuration errors within a single layer of a software system, this paper represents a first empirical study towards understanding the configuration options used by and shared between different layers in the WP multi-layer system, as well as potential links with comprehension problems of users. The results can then be used in a follow-up study on multi-layer configuration errors. In particular, we address two preliminary research questions to understand the evolution of configuration options across time, and two questions analyzing the usage of options across the studied layers:

RQ1: What is the proportion of usage of each configuration mechanism in each layer?

WP uses configurable constants and database configuration options equally, while WP plugins prefer (87%) configuration options stored in the database.

[2]https://wordpress.org/support/topic/plugin-contact-form-7-wont-connect-to-smtp-server
[3]https://wordpress.org/support/topic/plugin-nextgen-gallery-error-exceed-memory-limit

RQ2: How does configuration mechanism usage evolve across time in each layer?
Generally, the number of configuration options grows across time, especially when new features are added in a layer. Configuration options that are not used anymore are removed after a while.

RQ3: How many configuration options defined in lower layers are used by WP plugins?
On average, 1.49% to 9.49% of all WP database configuration options and 1.38% to 15.18% of all WP configurable constants are used by the plugins. Furthermore, 1.30 PHP configuration options are used by plugins, but only 0.40 ones are modified. These large numbers are confirmed by the StackOverflow and WP Exchange fora, where 12.19% of all plugin conversations and 9.49% of all WP conversations related to configuration mention multi-layer configuration options.

RQ4: How many plugins share the same configuration options of lower layers?
78.88% of all WP configurable constants and 85.16% of all WP database options are used by at least two plugins. For PHP, 52 PHP configuration options are used by at least two plugins. We found a strong correlation of up to 0.55 between the number of plugins using a configuration option and the number of fora conversations mentioning it.

The paper is organized as follows: section 8.2 presents the background and related work, and section 7.3 presents our methodology. Section 7.4 provides the results of our study, while section 8.6 discusses the threats to validity. Finally, section 8.7 concludes the paper and presents future work.

7.2 Background and Related Work

In this section, we provide background information about software configuration, multi-layer systems, and the WP ecosystem, and we discuss related work.

7.2.1 Software Configuration

Configuration is a mechanism to adapt software systems to a context or an infrastructure and is used to customize a system's behaviors. A configuration option is a pair consisting of an option name and its value, where the value has a specific type (typically boolean or categorical, but sometimes numeric or even a string). For example, a PHP configuration

option *file_uploads*, which is used to allow file upload or not, could have a value of either *On* or *Off*, while an option like *error_log* could have any string as its value.

Different configuration mechanisms exist, which typically differ in binding time and storage mechanisms. Values could be bound to configuration variables at compile-time, load-time (virtual machines) or run-time, while the values could be stored in the source code (macro constant or global variable), database, property files or in other supports.

A configuration error then is a set of configuration options that lead to unexpected behaviour, typically causing errors, even though the source code itself is correct. Yin et al. [215] provide an empirical study of a commercial storage system and four open source systems on configuration errors, and were able to classify 546 configuration errors into five major categories. Arshad et al. [43] provide a characterization of configuration problems for two Java EE application servers, GlassFish and JBoss, by analyzing 281 bugs-reports. Hubaux et al. [87] conduct two surveys respectively among Linux and eCos users to understand configuration challenges. Jin et al. [95] analyze two open source applications and one industrial application to quantify the challenges that configurability creates for software testing and debugging. Other studies focus on predicting configuration bugs. Using textual information in bug reports, Xia et al. [208] built a model to predict whether a bug is a configuration error or not.

Many studies have been conducted to resolve configuration errors. Keller et al. [101] proposed the tool *ConfErr* that aims at quantifying the resilience of a software system to configuration errors caused by spelling mistakes, structural errors, and semantic errors. Zhang et al. [217] built a tool to identify the root cause of a configuration error in Java programs. Zhang et al. [219] provide the tool *ConfSuggester*, which suggests the configuration option responsible for introducing a bug in a new version. The suggestion is generated based upon the control flow of a system. Elsner et al. [71] propose a framework to detect configuration inconsistencies. It allows a user to specify the possible inconsistencies in a software application, which will be combined with a model built from the configuration files to find the inconsistencies. Attariyan et al. [46] built the tool *ConfAid*, which aims at pointing out the root cause of configuration errors, again by analyzing the control flow. Tartler et al. [182] propose an approach to resolve the inconsistencies between the configuration model and its implementation in the Linux source code.

While the above research provides a set of configuration options that should be changed in order to fix a bug, Wang et al. [192] present an ordered set of configuration options to change in order to fix a configuration error, based on user feedback. Similarly, Xiong et al. [210] propose an approach that yields the configuration options to change and a range

of possible values. Lillack et al. [114] evaluate the tool *Lotrack*, which explains for each code fragment which load-time configuration options should be active for it to be executed. Nadi et al. [125] propose a static approach to extract and validate configuration constraints from C code, which would be hard for non-experts of a system to do manually. They also evaluate the approach's accuracy on four highly configurable open source systems. Rabkin et al. [147] use logs and traces to map each program point to the configuration options that could introduce an error. Jin et al. [94] built PrefFinder, using an NLP engine to provide the possible values of a configuration option. However, none of these related papers study multi-layer configurations.

While we do not study multi-layer configuration errors, we do a preliminary study of the prevalence of multi-layer configuration options. High prevalence would suggest that corresponding errors are likely and hence should be studied.

7.2.2 WP Ecosystem

Similar to Drupal and Joomla, WP is one of the most popular and powerful [138] Content Management Systems (CMS) for creating blogs. It powers more than 60 million websites, i.e., 61% of all websites created by a CMS[4], and 23.3% of all websites in existence[5]. One of the most important factors in WP' success is its variety of plugins. WP plugins (such as *NextGen*, and *Contact Form*) allow users to add to their websites any functionality that they can imagine, since a plugin basically consists of PHP scripts that can access any of the lower layers of the WP architecture. WP has thousands of plugins, together downloaded more than 748 million times[6].

WP typically runs on top of a multi-layer LAMP stack (Figure 8.1), consisting of a Linux operating system, Apache web server, MySQL relational database and PHP scripting language[7]. The vertical layout of Figure 8.1 shows how the different layers communicate with each other. The plugin layer communicates with the *Web framework* layer, which relies on the lower layer *Scripting language* that is used to connect to the *database* and *web server*. These elements in turn rely on the operation system, which hides the hardware complexity.

Some studies have focused on multi-language or multi-layer web applications. As PHP is a dynamic language used to create web pages, Nguyen et al. [130] propose a static analysis to find undefined variables and functions in all web pages generated by any HTML, JS, PHP,

[4]http://w3techs.com/technologies/overview/content_management/all/
[5]http://w3techs.com/technologies/details/cm-wordpress/all/all
[6]https://wordpress.org
[7]http://searchenterpriselinux.techtarget.com/definition/LAMP

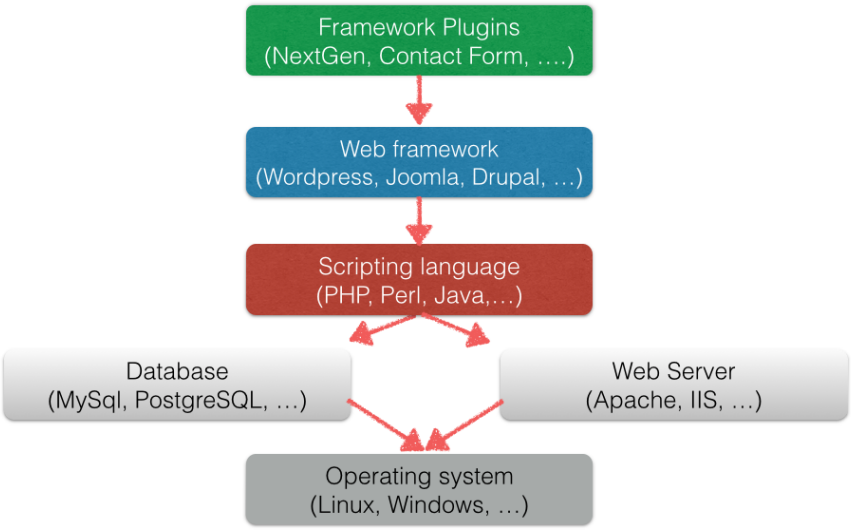


Figure 7.1 The layers of a typical WP installation. We focus on the configuration options of the top three layers.

or SQL script. Eshkevari et al. [73] study the problem of interference (conflicting entity names, hooks, database code, variables, and risky includes) between WP and 10 plugins, and propose an approach to resolve it. Nguyen et al. [128] elaborated a prototype PHP interpreter to detect WP plugin conflicts out of the large number of possible combinations (2^{50}) of activated WP plugins. They found that among all plugin combinations, 29% of WP statements and 89% of WP variables' values are shared. None of these papers study configuration options.

7.2.3 WP Configuration Mechanisms

WP and its plugins use two mechanisms for configuration. The first one consists of storing the configuration options in a database, while the second one consists of overriding PHP constants in the WP source code. We respectively refer to them by "database options" and "configurable constants".

Database options are stored in the table *wp_options*, while configurable constants are PHP constants that can be overridden by a user in a central configuration file *wp-config.php*. Every usage of the latter constants in the source code is preceded by an if-check (as shown in Figure 7.2) that checks whether the constant has been defined already. If not, it defines the constant with a default value. Since the *wp-config.php* file is loaded first by WP, any constant defined in that file by the user will have precedence over the default value.

```
if (! defined('FTP_FORCE')) define ('FTP_FORCE', true);
-------------------------------------------------------
$method = defined('FS_METHOD') ? FS_METHOD : false;
```

Figure 7.2 Two examples of configurable constants that can be redefined in *wp-config.php*

7.3 Approach

This section presents the methodology used to answer the research questions of the introduction.

7.3.1 Data Selection

Since RQ1 and RQ2 require manual analysis to complement the quantitative findings, we used a more focused data set for it ("Small Data Set"). For RQ3 and RQ4, we rely less on qualitative analysis and are able to study a larger-scale data set ("Large Data Set").

Small Data Set (RQ1 and RQ2)

For RQ1 and RQ2, we analyzed the source code of the WP layer (in the remainder of the text, we refer to "WP") and 15 WP plugins. The selection of plugins is based on the following two criteria:

- Criterion 1: Plugins should have a dedicated set of methods to extract configuration options from the database.
- Criterion 2: To avoid the need for intra- or inter-procedural data flow analysis, the parameter of the methods used to extract a plugin's configuration options should be specified as a literal.

While all plugins use the same methods to access the database options of the WP layer, each plugin can have its own methods to access its own configuration options. Since these methods are not known a priori (basically requires manual analysis), and these methods could change across time, the manual validation of criteria 1 and (especially) 2 took a substantial amount of time.

Based on these criteria, we randomly selected plugins for analysis from the popular plugins listed in the "add plugins" administrator page of a WP website, and obtained 15 plugins that satisfied the criteria (see Table 7.1). Although these plugins do not cover the top 15

Table 7.1 WP and plugins of the Small Data Set used in RQ1 and RQ2.

WP/Plugin	Versions (#)	# Downloads	Popularity Rank of Last Version
WP (platform itself)	1.5 - 4.0 (26)		
all-in-one-seo-pack	0.6.2.6 - 2.2.4.1 (232)	21.23M	2
updraftplus	0.7.4 - 1.9.5 (175)	1.70M	11
Google XML Sitemaps	2.5 - 4.0.8 (46)	16.20M	12
NextScripts	1.6.1 - 3.2.3 (11)	1.44M	15
wp-pagenavi	1 - 2.87 (23)	5.28M	27
Page Builder by SiteOrigin	1.2.10 - 2.0.3 (32)	1.15M	29
MailPoet	2.5.2 - 2.6.9 (26)	3.29M	33
Redirection	2.1.29 - 2.3.11 (19)	2.09M	34
The Events Calendar	1.5 - 3.9.1 (44)	1.35M	48
BBPress	2 - 2.5.4 (37)	1.66M	57
Download Manager	2.1.3 - 2.7.5 (7)	0.92M	58
broken-link-checker	0.1 - 1.10.4 (118)	3.14M	72
Captcha	2.12 - 4.0.7 (87)	2.42M	77
Flyzoo	1.4.2 - 1.4.5 (4)	0.32M	117
WP-Members	2.1.0 - 2.9.7 (49)	0.64M	132

most popular plugins, we can see that all plugins have at least 320,000 downloads, with a maximum of 21.23 million downloads for *all-in-one-seo-pack*. The plugins have between 4 and 232 versions.

Large Data Set (RQ3 and RQ4)

To analyze the interaction between different layers in the third and the fourth research questions, we use the last version of WP at the time of writing (4.0) and the 484 most popular plugins. This list of plugins can be obtained by any WP user from the administrator pages of a WP website. As we obtained in the first and second research questions the names of all WP configuration options, we just have to check the usage of those names in the plugins' source code. Furthermore, we found that for the methods used to access WP database options, the WP option name is passed as a literal argument in 98% of all the method calls. Therefore, we were able to use the 484 most popular plugins without limitations or specific criteria.

7.3.2 Identification of Configuration Options and Their Usage

Data Sources

We obtained the versions of WP and each plugin selected for the Small Data Set from the corresponding Subversion (SVN) repositories. In those repositories, WP and its plugins make all their versions accessible via SVN tags.

To obtain the 484 most popular plugins, we selected and installed the top 484 popular plugins (at the time of writing the paper) in our WP administrator environment.

Manual Identification of Access Methods:

Each plugin can have its own method to access the configuration options from the database. To understand these methods, we installed and activated each plugin, then looked at the database options added by the plugins of the Small Data Set, as well as how these configuration options are accessed in the source code of the plugin. Some plugins use the same methods as WP to access their own database configuration options, while others have their own methods. The number of methods differs from one plugin to another, and ranges from one to five.

As the plugins could change their methods across versions, we analysed the last version of each plugin, computed the number of configuration options for all plugins, then we checked manually for the previous versions if there was a big difference in the number of configuration options between two versions, which could be due to the modification of the methods to access database options. If so, we took the change in access methods into account.

7.3.3 Measuring The Proportion of Usage of Each Configuration Mechanism (RQ1/RQ2)

To obtain the configuration options used in the WP layer and the plugins layer, we performed the following two steps. First, to get the configurable constants of the plugins and WP, we scanned the source code for constants for which there is an if-check, as shown in Figure 7.2. For WP itself, we also scanned any constant already defined in *wp-config-sample.php*.

Second, to get the database configuration options of the plugins, we scanned the source code to find method calls to the methods that we know are being used by the plugin to access options (from our manual analysis in the previous section).

7.3.4 Measuring Direct Usage of Configuration Options (RQ3/RQ4)

To find the WP configuration options used by the plugins, and the PHP configuration options used by WP and the plugins, we used two approaches. The first approach measures direct usage of configuration options, i.e., textual occurrences of any of the known WP configurable constants (based on the list that we obtained in the two previous research questions) or explicit calls from a plugin or WP to obtain the value of a specific database option. This analysis uses regular expressions. The second approach (discussed in the next subsection) measures options accessed indirectly via nested method calls (e.g., a plugin calls a method of WP that accesses a configuration option).

To find accesses to PHP configuration options, we check the source code of WP and each plugin for calls to the methods *ini_get* or *ini_set*.

7.3.5 Measuring Indirect Usage of Configuration Options (RQ3/RQ4)

Given the gap in layers between the definition and usage of an option, RQ3 and RQ4 also need to deal with indirect usage of options. For example, instead of accessing a configuration option inside the source code of a plugin, the plugin might call a function of another plugin or maybe WP that accesses the configuration option.

We use the open source PHP-Parser[8] to build control flow graphs of all methods inside WP and each plugin, then perform a filtering of the control flow graphs. The filtering retains only statements that read or write configurable constants and database options of any of the three layers. Methods that are called but do not manipulate configuration options themselves are filtered out as well.

At the end of this filtering, the graphs contain all the data about configuration options and indirect access via method calls that we need. For each method, we can transitively follow the graph's edges to find all WP and PHP options that it reads or writes. For example, if the function x calls y and z, and y calls w, then by using the graph generated in this step, we return all the configuration options used in x, y, w, and z.

Note that direct usage of configuration options represents a lower bound of option usage for a given plugin or WP, with indirect usage corresponding to the worst case of additional options that could be accessed. We expect real usage to be closer to the lower bound, but included both bounds for completeness.

[8]https://github.com/cwi-swat/PHP-Parser

7.3.6 Measuring Configuration Options' Occurrences in Discussion Fora (RQ3/ RQ4)

To understand the actual impact of the findings of RQ3 (usage of options of deeper layers) and RQ4 (multiple plugins using the same options), we analyzed the StackOverflow and WP Exchange[9] fora respectively from July 2008 and November 2011 to March 2015, amounting to 7,259,572 and 46,509 conversations in total. We got the data of these fora in the form of xml files from the StackExchange archives[10].

We used StackOverflow and WP Exchange fora, because StackOverflow is one of the most popular fora for developers across all programming languages, while WP Exchange is a popular forum with WP developers. We did not analyze the WP support forum[11], because we could not distinguish conversations related to WP from those related to WP plugins.

For RQ3, we analyzed whether multi-layer configuration represents a real problem in practice, by measuring the percentage of conversations related to multi-layer configuration issues. For this, we searched for the names of WP and PHP configuration options inside forum conversations on plugins. We used the conversations' tags to distinguish between conversations related to WP and conversations related to plugins. A conversation related to WP plugins typically contains the keyword *"plugin"* like *"Wordpress-plugin"*.

For RQ4, we analyzed the same fora to find if there is a correlation between the number of plugins using a configuration option and the number of conversations about the option, which could give an indication about the existence of multi-layer configuration problems or confusion by users.

7.4 Results

In this section, we present for each research question the motivation, the approach used, and the results.

(RQ1) What is the proportion of usage of each configuration mechanism in each layer?

Motivation. We analyze in this research question the use of each configuration mechanism, with the goal of understanding which one is the most popular and hence needs closer analysis.

Approach. We used the approach of Section 7.3.3.

[9] http://wordpress.stackexchange.com
[10] https://archive.org/download/stackexchange
[11] https://wordpress.org/support/view/all-topics

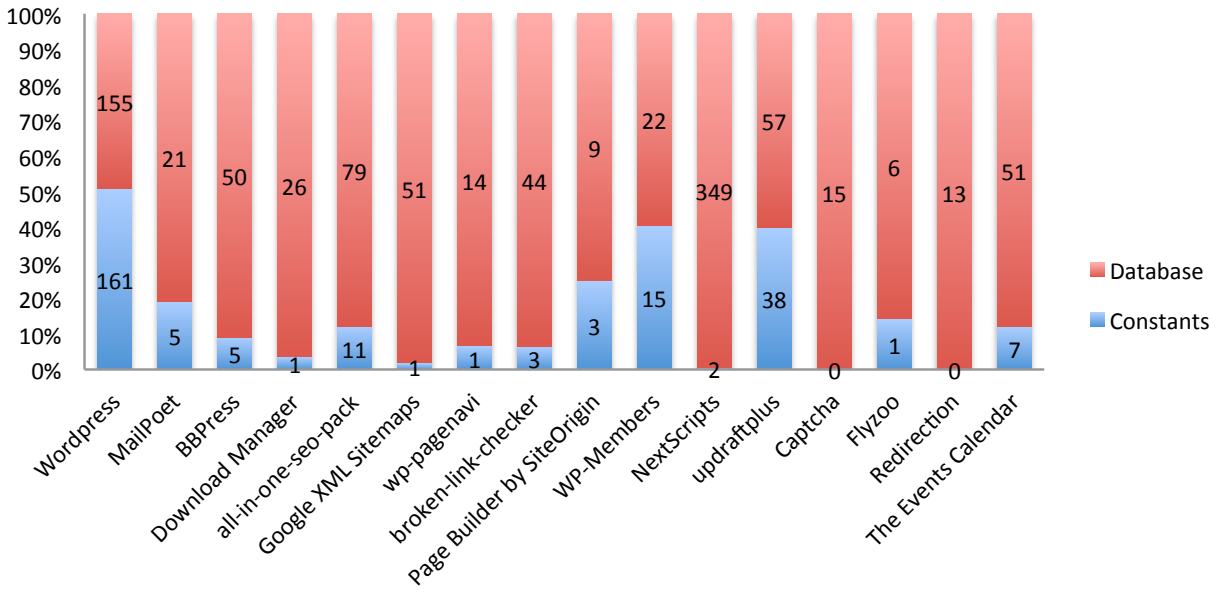


Figure 7.3 Distribution of the number of configurable constants and database options for WP and the 15 analyzed WP plugins.

Results. **The average number of configuration options across all plugins and WP is 76**. FlyZoo and Page Builder have the lowest number of options (less than 10), followed by *Redirection* and *Captcha*. *NextScripts* and WP have more than 300 options, followed by *updraftplus* and *all-in-one-seo-pack*.

On average 87% of a plugin's configuration options are stored in the database. Figure 7.3 shows the distribution of the number of configuration options across both mechanisms, for each plugin. It shows that WP uses 161 configurable constants as configuration option, and stores 155 configuration options in its database. In contrast, all WP plugins mostly use the database as a configuration mechanism, with an average of 87%, compared to 13% for configurable constants.

While the plugins use database options more than constants to configure their behavior, the percentage of usage of each mechanism differs from one plugin to another. There are some plugins where the configurable constants present 40% of all configuration options, such as *WP-Members* and *updraftplus*. Other plugins use the constants mechanism to configure around 20% and 25% of their configuration options, like *MailPoet* and *Page Builder by SiteOrigin*. The plugin *Flyzoo* uses constants as configuration mechanism for 14% of its configuration options, approximately the same as the plugin *The Events Calendar*. Finally, we have nine plugins where the configurable constants represent less than 10% of all configuration

options. The plugins *Captcha* and *Redirection* do not use configurable constants at all, they use only the database as a mechanism to store configuration options.

Discussion. To understand the dominance of database configuration options, we manually analyzed the documentation of the plugins and categorized the database options among different use case categories. We used the existing categorization of WPengineer.com[12] for configurable constants (Table 7.3), as inspiration for our categorization of database options.

Overall, the database configuration options are those that are shown in a plugin's public web interface, i.e., they are meant to be changed and customized by plugin users via the administrator pages. For example, we found a commit that removed a configuration option from the database[13], because it didn't have any UI anymore to change it. On the other hand, the constants, by definition, require changes to the code. While those constants conveniently can be overridden in the *wp-config.php* file, they require manual exploration of the source code to be detected and to understand the default value.

As presented in Table 7.2, we identified six database option categories. The first category represents the *"general configuration options"*, which refer to configurations used all over a website, such as the *blogname* or *siteurl*. The second category corresponds to *"writing"* options used to write the website pages and posts, such as the option *use_smilies*, which is a boolean variable used to display emoticons as graphic icons. The third category is *"reading"*, which presents all options related to displaying the posts, such as the number of posts per page (option *posts_per_page*). The fourth category corresponds to *"discussion"* options for the articles published on the website, such as *comment_registration*, which is a boolean variable used to decide whether commenters need to be registered. The fifth category corresponds to *"media"* options, i.e., the allowed dimensions of images. The last category corresponds to *"permalinks"*, which allows to customize the URL structure of blog posts and archives.

[12]http://wpengineer.com/2382/wordpress-constants-overview/
[13]https://core.trac.wordpress.org/changeset/27916

Table 7.2 Categories of database options.

Categories	Examples
General	siteurl, blogname, admin_email
Writing	use_smilies, mailserver_url
Reading	posts_per_page
Discussion	show_avatars, comment_registration
Media	thumbnail_size_w, large_size_h
Permalinks	permalink_structure, category_base

Table 7.3 Categories of WP configurable constants with examples by wpengineer.com [12].

Categories	Examples
General	AUTOSAVE_INTERVAL, WPLANG
Status	APP_REQUEST, DOING_AJAX
Path, dirs, and links	WP_LANG_DIR, ABSPATH
Database	DB_HOST, DB_USER
Multisite	MULTISITE, DOMAIN_CURRENT_SITE
Cache and script compressing	COMPRESS_SCRIPTS, ENFORCE_GZIP
Filesystem and connections	FTP_HOST, WP_PROXY_PORT
Themes	HEADER_IMAGE, TEMPLATEPATH
Debug	SCRIPT_DEBUG, WP_DEBUG
Security and cookies	COOKIE_DOMAIN, NONCE_KEY

On the other hand, if we compare to the categorization by WPengineer.com for configurable constants in WP (Table 7.3), configurable constants instead tend to manage more technical behavior, or correspond to configuration options that do not change regularly. For example, the general category contains technical options such as *DOING_AUTOSAVE*, which is used to specify whether *"WordPress is doing an autosave for posts"*[12]. There are other configurable constants that refer to development tasks, such as a category for debug mode, security, paths, directories, and links.

(RQ2) How does configuration mechanism usage evolve across time in each layer?

Motivation. The goal of this research question is to understand whether WP layers change the mechanism used to specify configuration options, or whether new options tend to prefer one mechanism or the other. It also sheds light on whether the number of configuration options plateaus early on, or whether the number keeps on increasing. The latter case would not only result into more options, but also potentially into more interactions between options, which could introduce more configuration errors.

Approach. For both configuration mechanisms, we use the same approach as in RQ1 for each plugin's version.

To analyze the results, we performed a number of activities. First, we used SLOCcount [203] to compute the number of lines of code of WP and its plugins to find if the increased number of options is related to large amounts of code (and hence features) being added. We also analyzed the names of the configuration options added in those versions with the goal of understanding whether or not the configuration options are related to new features. Finally,

we also used the tool DiffMerge[14] to compare the source code of a version introducing many configuration options with its predecessor.

Results. **For 60% of the plugins, the number of configuration options grows across time for both mechanisms.** As presented in Figure 7.5 (the full plots for all plugins can be found online[15]), the number of database options in the last version of a plugin is higher than in the first version, except for the plugin *Redirection* where the number remained the same. Similar observations hold for the configurable constants, except for those of the six following plugins (not shown): *MailPoet, BBPress, NextScripts, Captcha, Flyzoo*, and *Redirection*.

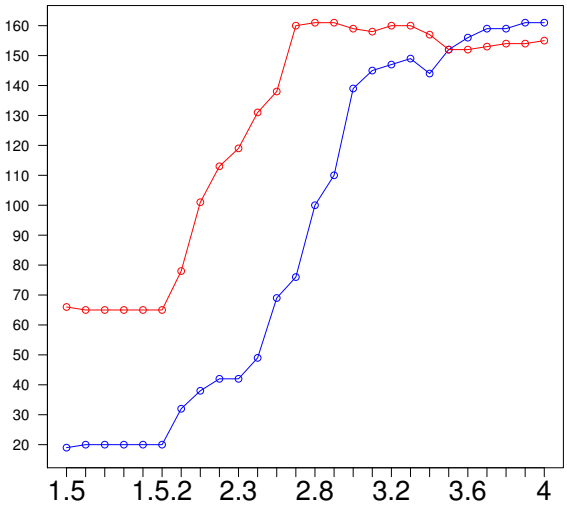
As we saw for RQ1, the number of configuration options stored in the database is higher than the number of configurable constants, and Figure 7.5 shows that this number remains higher across all versions of the plugins, except for WP, where the number of database options eventually dropped below the number of configurable constants.

Each plugin, except for *Redirection*, sees growth in its usage of at least one of the two configuration mechanisms. For WP, the number of options grows rapidly for both mechanisms, similar to the plugin *updraftplus* and *WP-Members*. For the other plugins, the number of configuration options stored in the database grows more rapidly than the number of configurable constants. The *Redirection* plugin sees no growth, and temporarily even lost one database configuration option between version *2.3.2* and version *2.3.5*.

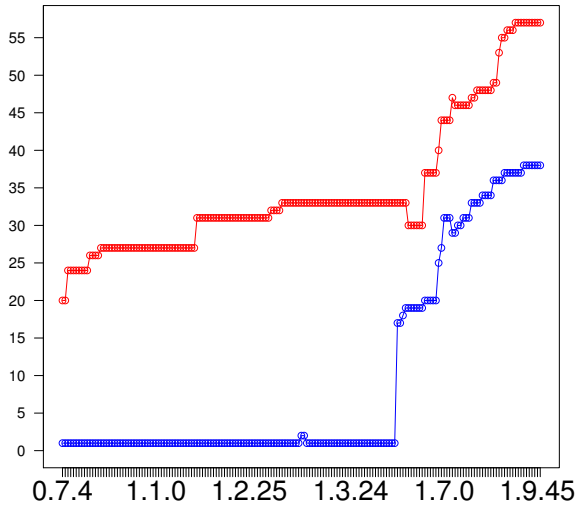
For some versions, we can observe that the number of configuration options has an important decrease. For example, the number of configuration options decreases between versions *3.4.1* and *4* of the plugin *Google XML Sitemaps* for database configuration options. The same happens between versions *1.1* and *1.2.2* of the plugin *broken-link-checker*.

Discussion. To understand the above findings better, we studied the Spearman correlation between the evolution of the number of configuration options and the size (i.e., number of source code lines) of WP and plugins versions. Table 7.4 shows that the correlation results are strong for WP and all plugins cases, except for *MailPoet* (moderate) and *Redirection* (moderate and negative). In the latter case, the number of configuration options is stable across all versions except two, one of which shows a decrease. We also plotted the number of options per line of code[16], which showed us that nine plugins have a decreasing trend, i.e., the size of these plugins increases more rapidly than their number of options. On the other hand, *Download Manager, NextScripts, Page Builder by SiteOrigin*, and *WP-Members*

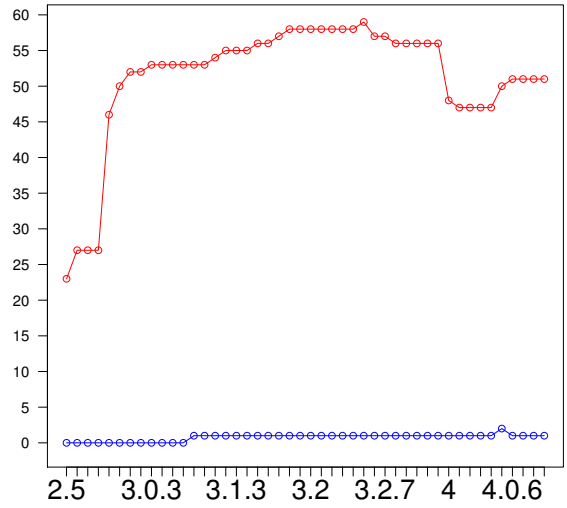
[14]http://www.diffmerge.net
[15]http://mcis.polymtl.ca/~msayagh/Paper/SCAM15/Figures/
[16]http://mcis.polymtl.ca/~msayagh/Paper/SCAM15/LOC_vs_Options/



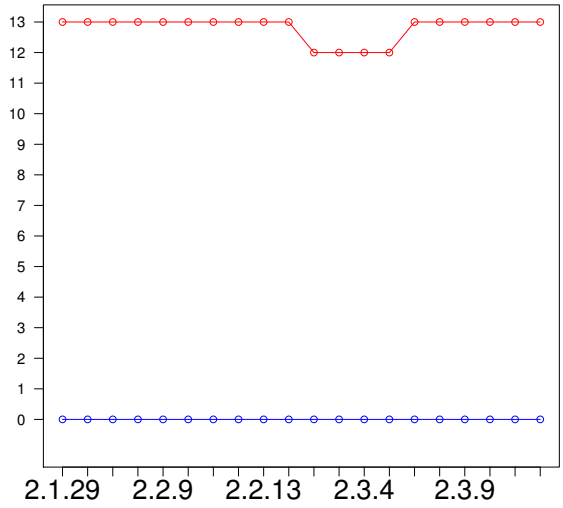
WP



Plugin: updraftplus



Plugin: Google XML Sitemaps



Plugin: Redirection

Figure 7.5 Evolution of the number of configuration options for both mechanisms across WP and the studied WP plugins versions, i.e., database (red) and configurable constants (blue), Our full results are online[15].

proportionally add more options than they increase in size and features, while *Flyzoo* and *Redirection* more or less retain a constant proportion.

As examples of these strong correlations, we found that between versions *2.0.3* and *2.1* of the plugin *BBPress*, its developers added 11 configuration options, since many features were added, represented by 7738 new lines of code. Just the addition of the component *bbConverter*[17] to the plugin introduced seven configuration options. In the plugin *all-in-one-seo-pack*, its developers added the following components: *Sitemap* and *Social Meta module*. Furthermore, some information was made configurable in the plugins *WP-Members* and *The Events Calendar*. For example, instead of hardcoding the different parts of an email like the body, or the mail footer, these now became configurable.

The growth of the number of configurable constants is due to making additional constants configurable or adding new components (containing configurable constants). By analyzing the difference between versions *2.8.10* (2 configurable constants) and *2.9.0* (16 configurable constants), we found that version *2.9.0* makes certain PHP constants of version *2.8.10* configurable, by testing if they are defined before their existing definitions (cf. Fig. 7.2). For the *updraftplus* plugin, we found that newly added components have 16 configurable constants.

There is one important case of decrease of configuration options between versions *3.4.1* and *4* of the plugin *Google XML Sitemaps*. Analysis of the source code showed the following comment: "*restores some default options which were not needed anymore in v4.*".

(RQ3) How many configuration options defined in lower layers are used by WP plugins?

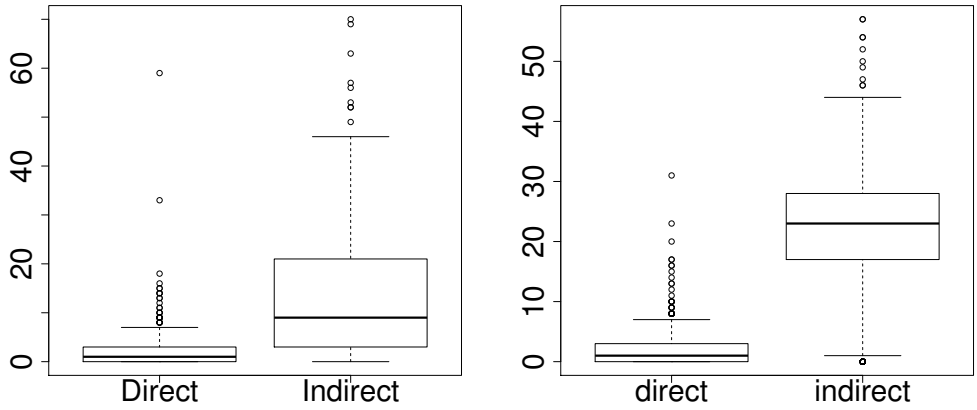
Motivation. Now that we better understand the scale and evolution of configuration option usage within layers, RQ3 analyzes the usage across different layers (see Figure 8.1). Such usage potentially can be error-prone, since configuration options of lower layers need to be defined in a different location than the plugins' options, and a change to the value of a lower layer option could impact multiple plugins at once. This RQ focuses on the plugins' usage of lower layer options, whereas RQ4 measures the amount of lower layer options used by more than one WP plugin.

Approach. We calculate the plugins' direct and indirect usage of WP and PHP options using the approach of Section 7.3.4 and 7.3.5. For database options of WP and for PHP options, we split up "usage" of an option into reading and writing (based on the names of access methods), whereas for constants we only consider reading (since by definition a constant can only be defined once). Note that for RQ3 and RQ4 we use the Large Data Set.

[17]https://wordpress.org/plugins/bbconverter/

Table 7.4 Correlation between number of configuration options and number of lines of code of WP and the analyzed plugins.

WP/Plugin	Correlation
Flyzoo	0.9933902
The Events Calendar	0.9884512
NextScripts	0.9860491
Download Manager	0.9814041
broken-link-checker	0.9762101
WP	0.9643059
updraftplus	0.9582439
BBPress	0.9447637
Captcha	0.937967
WP-Members	0.9121678
Google XML Sitemaps	0.8971884
all-in-one-seo-pack	0.8252223
Page Builder by SiteOrigin	0.728631
wp-pagenavi	0.7014725
MailPoet	0.491287
Redirection	-0.3714881



Database options Configurable constants

Figure 7.7 The number of WP options read by plugins.

Results. **Each plugin reads on average 1.49% to 9.49% of all WP database options, and 1.38% to 15.18% of all WP configurable constants.** While WP plugins read on average 2.32 WP database options directly, they read 13.73 options indirectly, corresponding respectively to 1.49% and 9.49% of the WP database options. Similarly, they read on average 2.41 WP configurable constants directly and 22.22 ones indirectly (i.e., 1.38% and 15.18%). Figure 7.7 confirms that the plugins read more configuration options indirectly than directly (both database options and constants). Note that the set of direct reads is not a subset of the set of indirect reads (although overlap is possible), since options in the latter category need to be read at least once in an indirect way to be considered as indirectly read.

The maximum number of WP database options read directly by one plugin is 59 (*JetPack* plugin), whereas for the indirectly read ones it is 57 (*Worker* plugin). The maximum number of WP configurable constants read directly by one plugin is 31 for *BuddyPress*. The plugin *JetPack* indirectly reads 70 configurable constants, which represents the highest number of configurable constants read indirectly.

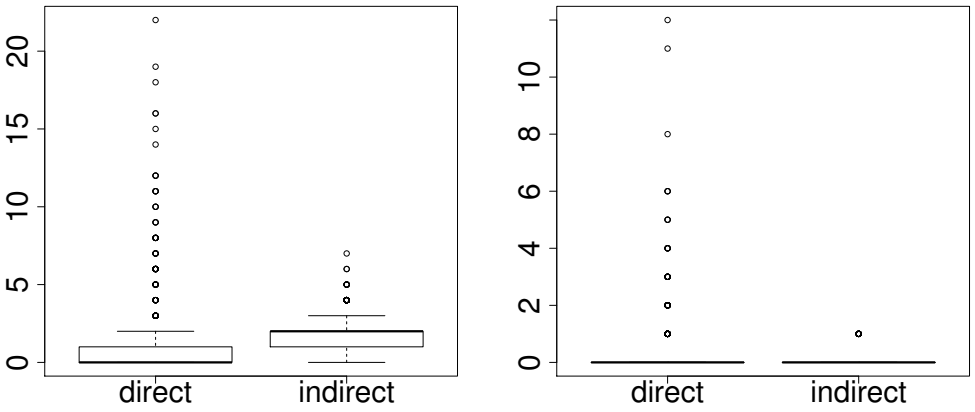
Each plugin directly writes on overage 0.11 WP database configuration options, and indirectly on average 0.32. While 441 plugins out of 484 do not directly write any configuration option, 35 plugins directly write one WP database option, five plugins write two WP database options, two plugins write three, and only one plugin writes five.

89 plugins (18% of all the plugins analyzed) indirectly write one or more WP database configuration options, where the maximum number of written WP configuration options is nine.

The plugins also read on average 1.30 PHP configuration options, and write 0.40 options, whereas WP reads 20 PHP options, and writes 8 options. Figure 7.9 shows that the plugins read the PHP options more indirectly than directly. It also shows that the number of writes is low (median of 0).

Discussion. Database options and configurable constants are mostly being read by their own plugins. This can be seen by the fact that the plugins use just 1.42% of the WP database options directly, and just 1.49% of the configurable constants. Moreover, few PHP configuration options are read by plugins, just 1.30 on average, and few of them (0.40) are modified. The use of PHP configuration options by WP is also negligable.

Although this seems good news, the number of configuration options used indirectly is significantly higher than the number of options used directly. Hence, the modification of one configuration option could impact the behavior of many plugins at once, possibly without developers being aware (since the dependencies are indirect). Figure 7.7 also shows that con-



Reading Writing

Figure 7.9 The number of PHP options used by plugins.

figurable constants are used more indirectly than database options. We study this in more detail in the next RQ.

To better understand the degree to which cross-layer configuration option usage could cause problems to users in terms of understanding and maybe errors, we analyzed the StackOverflow and WP Exchange fora. Out of 60,502 StackOverflow conversations related to only WP (not plugins), 5,907 (9%) are related to configuration issues, and 816 of these 5,907 conversations (13%) are related to PHP configuration options. Stackoverflow also contains 8,417 conversations related to WP plugins, where 679 conversations (8%) contain at least one WP or PHP configuration option, i.e., are related to multi-layer configuration issues. The WP Exchange forum[9] contains 9,756 conversations related to WP configuration issues out of 46,509 (21%), where 504 (5%) are related to PHP configuration options. From the 8,426 conversations related to plugins in the WP Exchange forum[9], 1,375 conversations (16%) are related to WP or PHP configuration.

Although the percentages of plugin conversations related to multi-layer configuration (8% and 5%) seem low, it is important to keep in mind that we do not know the number of plugin conversations talking about configuration in general (as the plugins' options follow different naming conventions). Since this number will be much lower than 8,417 or 8,426, the percentage of *configuration* discussions that consider options across layers will be much higher than 8% or 5%.

Therefore, an important percentage of conversations in both fora is related to multi-layer configuration issues, which suggests that it is an important issue for WP users.

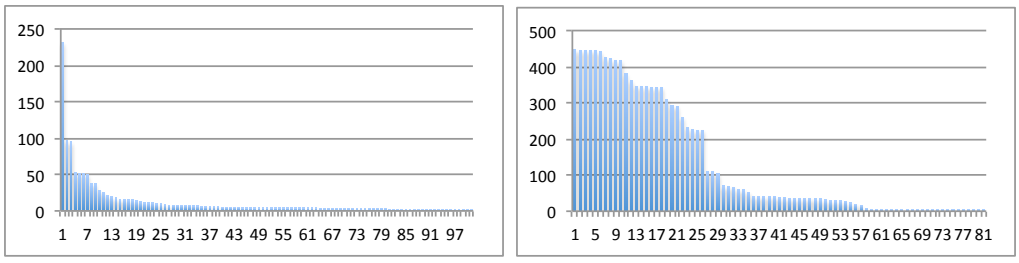
(RQ4) How many plugins share the same configuration options of lower layers?

Motivation. In this research question, we analyze interference between plugins caused by dependency on a common configuration option of WP or PHP. Such interference could indicate risky configuration options of lower layers that might impact many plugins at once.

Approach. Similar to the previous research question, we measure both direct and indirect configuration option usage. Since here we are interested in any usage of configuration options by one or more plugins, we do not distinguish between reading and modification of options, but merge both into "usage". For our study of Spearman correlations of forum conversations, we merged the data of both fora into one.

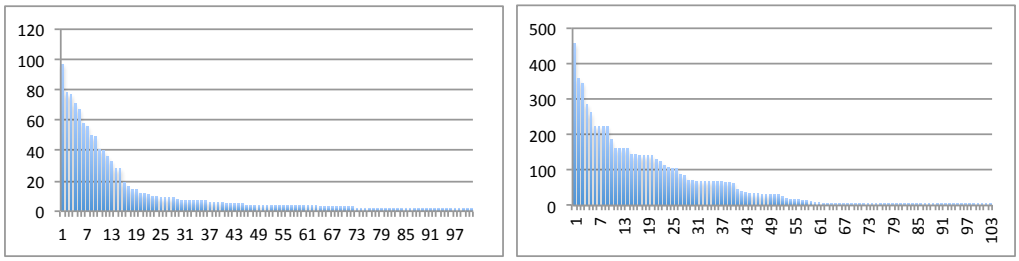
Results. **78.88% of all WP configurable constants and 85.16% of all WP database options are used by at least two plugins.** In Figure 7.11, 25 out of 161 WP configurable constants are used directly by more than 10 plugins, while 75 configurable constants are used indirectly by more than 16 plugins (29 even by more than 104 plugins). The highest number that we found were the 231 different plugins that directly use the configurable constant *ABSPATH*. For indirect usage, *WP_DEBUG* is used by 447 different plugins out of the 484 plugins in the Large Data Set, and the *VHOST*, *MULTISITE*, and *SUNRISE* options are used by 445 different plugins. At the other extreme, 21 configurable constants are used directly by just two different plugins, and five are used indirectly by two different plugins.

Turning now to WP database options, Figure 7.13 shows how 24 out of 155 database options are used directly by more than 10 plugins, whereas 59 are used indirectly by more than 10 plugins. The option used directly by the highest number of plugins is *active_plugins*, which is



direct indirect

Figure 7.11 The number of plugins (Y axis) using a given configurable constant (ordered on the X axis).

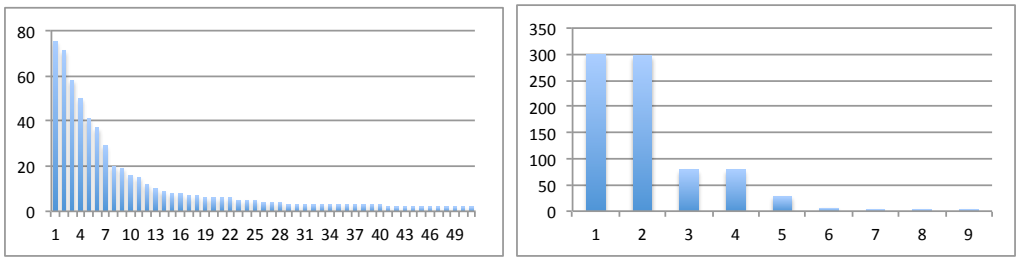


direct indirect

Figure 7.13 The number of plugins (Y-axis) sharing the same database configuration option (ordered on the X axis).

used by 97 different plugins out of 484 plugins, and the second most important configuration option is *siteurl* (used by 78 plugins). The option *blog_charset* is used indirectly by the highest number of plugins (458 different plugins). At the other extreme, 28 WP database configuration options are used directly and seven ones are used indirectly by only two different plugins.

52 PHP configuration options are used by at least two different plugins. In Figure 7.15, 13 PHP configuration options are used directly by more than 10 plugins, while five PHP configuration options are used indirectly (across WP functions) by more than 28 different plugins. The PHP configuration *memory_limit* is used directly by 75 different plugins, *safe_mode* is used by 71 plugins. However, the most used options are *arg_separator.output* and *mbstring.func_overload*, which are used respectively by 300 and 297 different plugins.



direct indirect

Figure 7.15 The number of plugins using the same PHP configuration option (ordered on the X axis).

There are also three PHP configuration options used by just three different plugins, and nine PHP configuration options that are used indirectly from the WP source code.

Discussion. As 101 (62%) out of 161 of WP configurable constants are used directly and 81 (50%) are used indirectly by at least two different plugins, the modification of a configurable constant inside the WP layer might impact many other plugins indirectly. For example, the configurable constant *ABSPATH* is used directly by 47% of the plugins studied, and indirectly by 78%. If its value would become corrupt, it could impact the behavior of at least 78% of the plugins.

Similarly, there are some WP configuration options stored in the database that are used by many plugins, which means that their modification could impact the plugins' behavior as well. For example, *blog_charset* is used indirectly by 94% of the plugins studied. An error with this option could impact what the plugins display on the screen. The above of course holds for the PHP configuration options as well, which come from an even deeper layer.

We found that there is a weak to moderate correlation (between 0.22 and 0.55) between the number of times a WP configuration option is being used directly by plugins and the number of conversations discussing the option, as shown in Table 7.5. The correlation results for indirect usage of WP configurable constants and WP database options are weaker. All correlations are positive, hence the more plugins use an option of a deeper layer, the more discussions there are about the option, hence the more information people require about it. This suggests that the reuse of configuration options of deeper layers can pose problems for WP users and hence requires more research.

Table 7.5 The correlation between the number of plugins using an option and the number of conversations mentioning it.

Configuration options of	Case	Correlation
WP configurable constants	Direct use	0.5533475
WP configurable constants	indirect use	0.2268492
WP Database options	Direct use	0.3643119
WP Database options	indirect use	0.2795615
PHP configuration options	Direct use	0.5593974
PHP configuration options	Indirect use	0.4427347

7.5 Threats to Validity

Regarding threats to external validity, since we analyzed only the WP ecosystem, and within this ecosystem only a limited amount of plugins, we cannot generalize the results to other systems. However, our results provide a first large analysis of the use of configuration options in multi-layer systems, since in total we considered 484 plugins, 15 of which were manually analyzed across time. In future work, we plan to analyze other multi-layer systems, in the same as well as other domains.

Regarding threats to internal validity, we analyzed only 15 plugins for the first and the second research question due to the manual analysis required, especially due to the criterion for literals in the method calls used to access the plugins' database options for RQ1 and RQ2, and the need to manually find these methods. However, these plugins are all popular plugins from a variety of organizations and domains.

7.6 Conclusion

Multi-layer systems like WP have a potential for configuration errors due to interference between configuration options in different layers. As a first step towards analyzing such errors, this paper performed an empirical study on the prevalence of multi-layer configuration options in WP, WP plugins and the PHP system. We found that except for WP itself, WP plugins prefer storing configuration options in a database, in order to make them easily available to the end user for configuration. Furthermore, configuration options and usage evolve across time, especially when new features are added. Across layers, we found that each plugin uses on average 1.30 PHP configuration options and modifies 0.40 options, while it uses on average 1.49% to 9.49% of all WP database options and more than 1.38% to 15.18% of all WP configurable constants. Furthermore, 78.88% of all WP configurable constants and 85.16% of all WP database options are used by at least two plugins at the same time, which can be between two and 447 plugins for configurable constants, two and 458 plugins for WP database configuration options, and between two and 300 plugins for PHP configuration options.

Finally, there is more indirect use of configuration options than direct use, which could make the detection and fixing of configuration errors more difficult. We indeed found initial evidence of this potential through the relatively high percentage of conversations in Stackoverflow and WP Exchange fora talking about options of deeper layers. Hence, we would suggest Wordpress to provide a mechanism to warn plugin developers or users for the impact

of cross-layer configuration modifications. Future work needs to build on these results to help users detect and fix configuration problems across multiple layers.

CHAPTER 8 ARTICLE 5: ON CROSS-STACK CONFIGURATION ERRORS

Mohammed Sayagh, Noureddine Kerzazi, Bram Adams
Published in the 39th International Conference on Software Engineering (ICSE)

Abstract: Today's web applications are deployed on powerful software stacks such as MEAN (JavaScript) or LAMP (PHP), which consist of multiple layers such as an operating system, web server, database, execution engine and application framework, each of which provide resources to the layer just above it. These powerful software stacks unfortunately are plagued by so-called cross-stack configuration errors (CsCEs), where a higher layer in the stack suddenly starts to behave incorrectly or even crash due to incorrect configuration choices in lower layers. Due to differences in programming languages and lack of explicit links between configuration options of different layers, sysadmins and developers have a hard time identifying the cause of a CsCE, which is why this paper (1) performs a qualitative analysis of 1,082 configuration errors to understand the impact, effort and complexity of dealing with CsCEs, then (2) proposes a modular approach that plugs existing source code analysis (slicing) techniques, in order to recommend the culprit configuration option. Empirical evaluation of this approach on 36 real CsCEs of the top 3 LAMP stack layers shows that our approach reports the misconfigured option with an average rank of 2.18 for 32 of the CsCEs, and takes only few minutes, making it practically useful.

8.1 Introduction

Every web app requires a so-called "software stack" to provide the computation and storage resources that it needs. For example, a web app would not be accessible without a web server. Moreover, a web app requires a database to store its state, and some kind of execution engine within which computations can be run. Hence, a web app requires a large set of services, each of which is served by a separate layer, together forming a stack of services consuming each other's resources. One popular software stack is the so-called LAMP stack (Figure 8.1) [22], consisting of Linux (operating system; OS) [23], Apache (web server) [6], MySQL (database) [26] and PHP (execution engine) [28] layers for deploying web apps such as Wordpress [33] (WP) or Drupal [16] (DR). Other common stacks are the J2EE [20] and MEAN stacks [25].

Once a web app is deployed, the behaviour of the stack can further be adapted to a particular platform by changing the layers' configuration options. Such options are basically a set of <key,value> pairs, in which the key represents an option name and the value a user's desired choice for that option. These pairs typically are stored in dedicated configuration stores (files, databases, ...), and can change the behaviour of a system without re-compilation. For example, one can configure the database server to limit the number of connections by using the database option "max_connections", while one can also configure the PHP-interpreter to limit the execution time of a script by the option "max_execution_time".

Despite this flexibility, assigning a wrong value to a configuration option could lead the configured stack to behave incorrectly or even to crash. Although, technically, one only needs to change the value of a configuration option to fix it, finding the correct option(s) to change and the correct value is the topic of ongoing research [71, 113, 125, 182, 192, 211, 217, 219]. Moreover, configuration errors have a severe impact. Studies [215] have shown that configuration-related errors can account for 27% of all customer-support cases in industrial contexts, while a well-known Google engineer prioritized them as one of the top directions for future research major problems.

While resolving misconfiguration errors related to a single layer of a software stack is difficult, such errors can span across multiple stack layers, which could make them even harder to troubleshoot and resolve [185]. Indeed, each stack layer has its own configuration options and programming language [95], and some features could be managed by different options. For example, the memory size a script is able to use can be changed in the WP layer by changing the option "WP_MEMORY_LIMIT", in the PHP interpreter layer by "memory_limit", and in the web server layer by "php_value memory_limit". Such configuration choices could contradict each other and hence confuse end users. In the context of the WP LAMP stack, a notorious example of a Cross-stack Configuration Error (CsCE) was the inability of users of the NextGEN Gallery plugin to upload an image due to a memory misconfiguration in the lower PHP interpreter layer.

This paper empirically studies the characteristics of CsCEs, then proposes and empirically evaluates a novel modular algorithm that leverages existing code slicing approaches. The algorithm analyzes configuration options and their code dependencies across multiple layers of a stack to recommend the configuration option most likely causing a CsCE. We make the following contributions:

- A large, qualitative study on 1,082 configuration errors obtained from 3 online discussion forums to understand the impact of CsCEs, the effort required to resolve such errors, and the complexity of CsCE fixes.

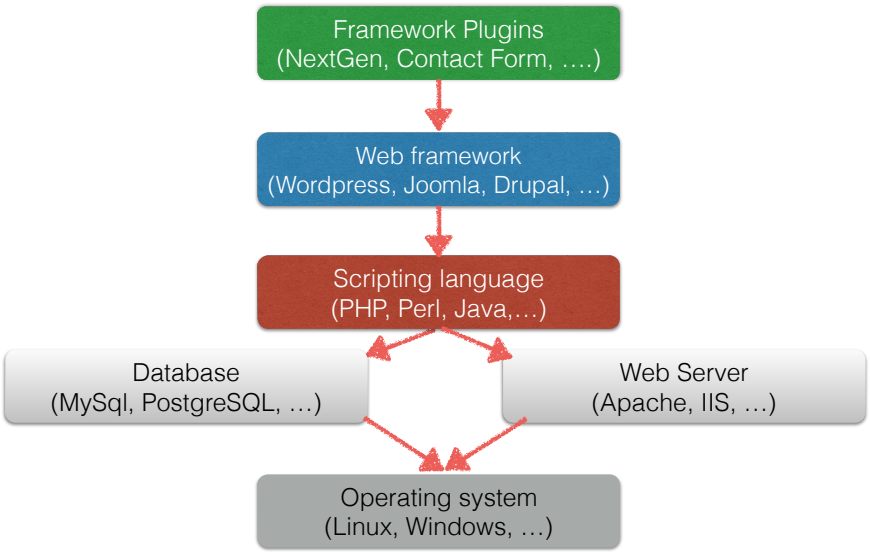


Figure 8.1 *Architecture of a LAMP stack.*

- A modular algorithm that allows to plug in existing code slicing techniques to recommend the configuration options that are most likely responsible for a CsCE.
- A large empirical evaluation of the algorithm on 36 real CsCEs in WP, DR, and 7 WP plugins, showing that our approach reports the culprit option within few minutes with a rank average of 2.18 for 32 out of the 36 cases.
- A public dataset of 36 evaluated CsCEs [160].

8.2 Background and Related Work

This section presents background and related work about software stacks, configuration errors, and CsCEs.

8.2.1 Software Stacks

Definition. We define a software stack as an acyclic graph where nodes represent layers and an edge connects a layer to another layer whose resources it requires. Figure 8.1 shows the LAMP stack and mentions different components possibly existing within each layer. Here, WP plugins extend the functionality of the WP framework, which needs resources from PHP for computation (e.g., standard library), which relies on MySQL and Apache for database

and web access, etc. Note that stack dependencies model resource usage, not the order in which an HTTP request is being served by a web server.

LAMP Stack. Even though the problem of CsCEs and our approach apply to any software stack that fits our definition (e.g., MEAN or J2EE), this paper uses the LAMP stack as example, in particular with the WP and DR web apps. WP and DR are two of the world's most successful Content Management Systems. WP 4.6 had 5.3 million downloads, 29 thousand plugins and serves 74.6 million web sites, while DR 5.x–8.x together have around 1.2 million downloads, 35.2 thousand plugins and serve over 1 million websites. This success is largely due to the variety of themes and plugins, which are PHP scripts that access the WP/DR layers (PHP) to extend basic functionalities, for example easier image uploads, extra widgets or interfacing with other web apps.

Existing Work. Many research efforts focus on WP as case study. In prior work [162], we found that each WP plugin is using up to 15.18% of the WP layer's configuration options, while 82% of all WP options are used by at least two different WP plugins, suggesting a large risk of CsCEs. Nguyen et al. [128] verify plugin conflicts by testing all possible combinations of enabled WP plugins at once. They also [130] proposed an approach to detect undefined variables and functions across all possible instances of a web page. Eshkevari et al. [73] proposed an approach to detect interference problems like conflicts between entity names, or between generated client codes between WP and its plugins.

8.2.2 Single-layer Configuration Errors

Definition. A configuration error is an incorrect system behavior due to a bad value assigned to an option. It typically has as symptom an error message generated by the source code. A single-layer configuration error is a configuration error in a software stack where the symptom and misconfigured option are known to belong to the same stack layer.

Existing work. Many researchers focused on understanding configuration errors. Yin et al. [215] classified 546 misconfiguration errors from four open source systems and one commercial software system into different categories to understand the different causes of such errors. Jin et al. [95] conducted an empirical study to understand the challenges that configuration introduces for testing and debugging. Hubaux et al. [87] conducted an empirical study to find the challenges of configuration across Linux and eCos users, and found that better configuration support is required. Arshad et al. [43] analyzed 281 bugs of two Java EE application servers, in order to characterize configuration errors. Together with our prior work [162], these papers found that configuration errors are an important problem and are hard to debug. However, as reported by Xu et al. [185], none of these papers focus on CsCEs.

Two major strategies have been used to analyze or test the configurations of a system. Sampling algorithms [117] try to select the most representative configurations for analysis or testing using conventional analysis/test techniques. Conversely, variability-aware approaches [112, 184] aim to analyze all configurations at once, by making analysis or testing tools configuration-aware (e.g., aware that a particular line is only executed for a specific option value).

Several debugging approaches for single-layer configuration errors exist. Zhang et al. [217] resolved misconfiguration errors in Java programs by comparing erroneous program executions with a pre-built database of correct execution profiles. Later [219], they used historical information to identify which configuration option is introducing a bug in a new version. Dong et al. [68] instead used static slicing, while Attariyan et al. [46] use control flow analysis. Wang et al. [192] rank the reported culprit options based on user feedback. Xiong et al. [211] used constraint models to not only identify misconfigured configuration errors, but also propose a correct value.

8.2.3 Cross-stack Configuration Errors

Definition. In the most narrow sense, a CsCE is a configuration error whose symptoms are reported in a stack layer without any direct link (access) to the culprit option. For example, a permission problem in an option of the PHP interpreter in Figure 8.1 could yield a "file cannot be uploaded" error in the WP layer, without any read or write of the culprit option in the latter layer. The definition of CsCE could be broadened to cases where the layer with the error symptom (a) reads the misconfigured option, but the option and the incorrect value assignment still belong to a different layer; or (b) both reads and assigns the misconfigured option, but the user is misled to think that the symptom is generated by a different layer. The latter case is very common, as web or database error messages often are returned as is by the top stack layer. Although we target the narrow definition of CsCE, our approach can also help with the two broad interpretations.

Existing work. To the best of our knowledge, no existing approach resolves CsCEs [185]. One alternative could be to use the single-layer approaches discussed in the previous section.

Even though sampling-based approaches cannot guarantee to detect all errors [117], they could be used to prevent certain CsCEs, complementing debugging techniques. Yet, thus far, they have not been evaluated on run-time configuration [112,117], using preprocessor macros and conditions to store/check option values instead of regular variables and if-conditions. Furthermore, each stack layer brings its own options, of string type (not just boolean), and can be implemented using different technologies (e.g., MySQL vs. PostgreSQL), yielding

a significantly larger search space to sample from. Variability-aware approaches are not immune to this search space explosion either, as they require customization to the specific configuration (variability) mechanisms and layers used.

Many approaches for debugging single-layer configuration errors require additional input that is hard to obtain in a software stack. For example, several [71, 182, 211] require configuration constraint models as input, i.e., the allowed combinations of option values. Approaches to generate such models have only been applied on single-layer systems using the C preprocessor [125] or other configuration conventions [71, 182, 213]. Similarly, other approaches require data of a correct version of the system [217, 219]. This is more difficult to obtain in a software stack context, as it requires to have access to all configuration data for an identical stack setup (same layer technologies and versions), which also increases the volume of data to process. Our approach does not require any oracle or model and, hence, is able to scale to CsCEs.

The single-layer approach closest to our work is the static slicing-based one of Dong et al. [68]. Our modular algorithm can plug in a single-layer slicing-based technique such as this to make it layer dependency-aware (and combine it with similar techniques for different layers). Finally, Attariyan et al. [46] do not use slicing, but dynamically explore and replay condition branches to avoid the error and hence find the culprit option. Replaying a large number of branches can be costly in terms of time, and this only aggravates for multi-layer systems, which have significantly more branches and options.

Recently, research has started focusing on understanding software systems that contain multiple programming languages, of which software stacks form a subset. For example, Kochhar et al. [107] found a correlation between the number of programming languages in a system and the system's overall quality. One such quality issue, studied by Nguyen et al. [129], is consistency between variables shared between different languages (in particular JS, HTML and SQL). They developed a custom slicer for PHP web apps that combines traditional program analysis with symbolic execution and abstract interpretation in order to handle JS, HTML and SQL code embedded in PHP code. While they did not address nor evaluate their approach for configuration problems, a restricted version of the slicer (focusing only on configuration variables) could be integrated into our generic approach to handle other stack layers and technologies than those it was built for.

8.3 Qualitative Analysis

To gain an in-depth understanding of the impact of CsCEs, the effort required to fix them and the complexity of the obtained fixes, this section presents the results of a qualitative study on 2,387 forum threads from 3 online Q&A platforms.

8.3.1 Methodology

Data sources. Given the vast variety of software stacks, we focused our qualitative analysis on the popular LAMP software stack (Figure 8.1), in particular on configuration errors in the Apache, MySQL, PHP and Wordpress (WP)/Drupal (DR) application layers. Due to time limitations, we did not consider Linux-related errors, nor errors related to WP/DR plugins.

Basically, we conducted a "top-down" and a "bottom-up" analysis of cross-stack and single-layer errors. The top-down study analyzes both kinds of errors in the WP/DR layers, in the context of a LAMP stack. The bottom-up approach instead focuses on configuration errors caused by the PHP, Apache, or MySQL layer, regardless of which layers are running on top of them (i.e., not necessarily a LAMP stack). We used the study design outlined in Table 8.2 on the data sources of Table 8.1, with columns 2 and 3 of Table 8.2 corresponding to the top-down analysis, and 4 to 6 to the bottom-up analysis.

Amongst the studied data sources, StackOverflow is a general Q&A site covering a wide range of topics for the general public, users and developers. We selected only those questions that were marked as solved and tagged with "wordpress" or "drupal", then filtered the resulting questions by searching for discussion comments (not code blocks) mentioning the names of configuration options of Wordpress/Drupal (single-layer candidate errors; SO) or PHP/MySQL/Apache (cross-stack candidate errors; SO-PHP/MySQL/Apache). We obtained those names from the documentation of the corresponding layers. For Apache, we only retained discussions mentioning the filename "httpd.conf" to reduce false positives.

StackExchange has subcommunities dedicated to Wordpress and Drupal users and developers. We used the same approach as above, yielding the single-layer (STE) and cross-stack configuration error candidates (STE-PHP/MySQL/Apache) for the analyzed layers. Finally, ServerFault is a Q&A site targeted by system administrators. We searched for questions mentioning options of PHP, MySQL or Apache. As mentioned earlier, ServerFault is not limited to LAMP, hence our bottom-up analysis is able to find options causing CsCEs in any stack.

Approach. Two human raters (first two authors) independently analyzed the selected questions and their discussions on the three Q&A sites to determine the values of 5 numeric, 3

Table 8.1 *Qualitative data source statistics.*

Q&A platform	Time Period	#Threads	#Config. Errors
Stack Overflow	Aug '09-Jun '16	997	359
Stack Exchange	Sep '10-Feb '16	605	211
Server Fault	May '09-Feb '16	756	483

Table 8.2 *Overview of the five layers and three data sources analyzed for the qualitative study. 'SO' stands for StackOverflow, 'STE' for StackExchange and 'SF' for ServerFault.*

	WP	Drupal	PHP	Apache	MySQL
single	SO/STE	SO/STE	SF	SF	SF
cross	SO/STE-PHP/...	SO/STE-PHP/...	SF	SF	SF
	MySQL/Apache	MySQL/Apache			

boolean and 7 textual characteristics. For the textual characteristics, each rater could assign arbitrary tags such as "production environment" or "conflicting option". Since this resulted in a large set of tags for these characteristics, and in order to resolve disagreement between raters, they performed card sorting [155] for each textual characteristic. This allowed to cluster tags into either fewer or broader categories, effectively turning these textual characteristics into nominal data. They then revisited the discussions, replacing their initial tags by the corresponding nominal cluster names.

Of the 2,387 studied discussions, 44.7% (1,082) were related to configuration errors (Table 8.1). Figure 8.2 shows for each of the 5 studied layers of Table 8.2 a comparison between the number of analyzed configuration errors that are single-layer versus those that are cross-stack. Note that this figure does not allow comparison between layers. For Drupal and PHP, we found substantially more CsCEs than single-layer errors, while for Apache we found the opposite. Given the imbalanced data for Drupal, PHP and Apache, we decided to aggregate the data of the five layers, obtaining 539 single layer and 543 CsCEs.

8.3.2 Impact of Cross-stack Configuration Errors

We found a statistically **significant difference in the distribution of impact** for single-layer and CsCEs (χ^2 test; p-value $< 2.2e^{-16}$ with $\alpha = 0.01$). As shown in Figure 8.3, CsCEs are more severe compared to single-layer errors in terms of the percentage of crashes occurring (47% of all CsCEs compared to 29%), while they have approximately the same percentage of hangs (23% vs. 20%). Our card sort analysis shows that cross-stack configuration crashes

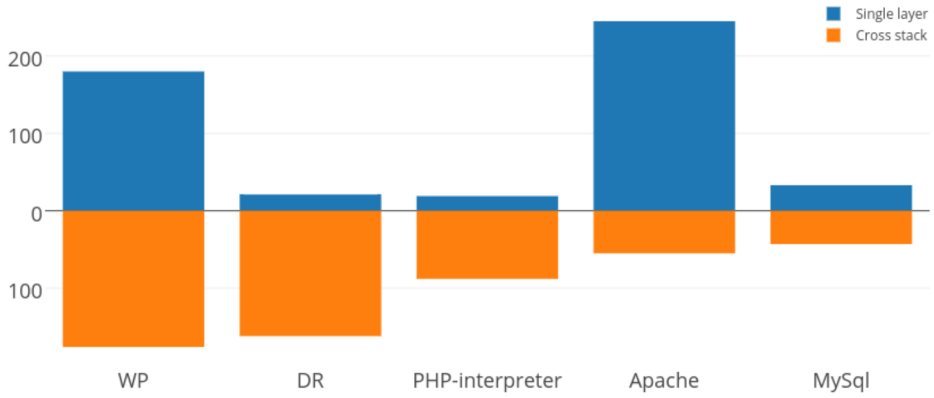


Figure 8.2 *Difference between the number of single-layer errors and CsCEs for each case study.*

typically are related to lower-layer options that control the stack's capacity, like the memory size ("memory_limit") or execution time ("max_execution_time") allowed for a script. Surpassing these limits ends up with a crash. On the other hand, single-layer errors are more related to user access permissions than CsCEs, but such errors do not tend to crash the system.

Since errors in the production environment are more severe, and **at least half of the single-layer and CsCEs occurred in production**, we refined the results of Figure 8.3 to production errors only. We found that CsCEs exhibit a more severe impact compared to single-layer errors, **even in production** (χ^2; p-value of $1.106e^{-11}$). Again, the vast majority of CsCEs were crashes (45%), a percentage that is much higher compared to single-layer errors (24%).

The reason for this, besides the mistake of using different environments for testing and production, is the lack of testing at scale before production. For example, in one case [24] the "max_input_vars" option caused a CsCE where adding more than 90 menu items to a WP site crashed the system. The site had never been tested with more than a handful menu items. Single-layer production crashes are more related to misconfigured URLs and paths to lower layers, which break when for example another database or web layer is installed.

Figure 8.4 shows that users face the majority of single-layer configuration errors just after setting up their stack (31%), or during application maintenance (26%), for example when a new plugin is installed or theme is changed. These problems have a relatively low impact as they can be resolved before release or while the system is undergoing maintenance. However, **CsCEs frequently occur during DevOps activities** (26% vs. 14%) such as running

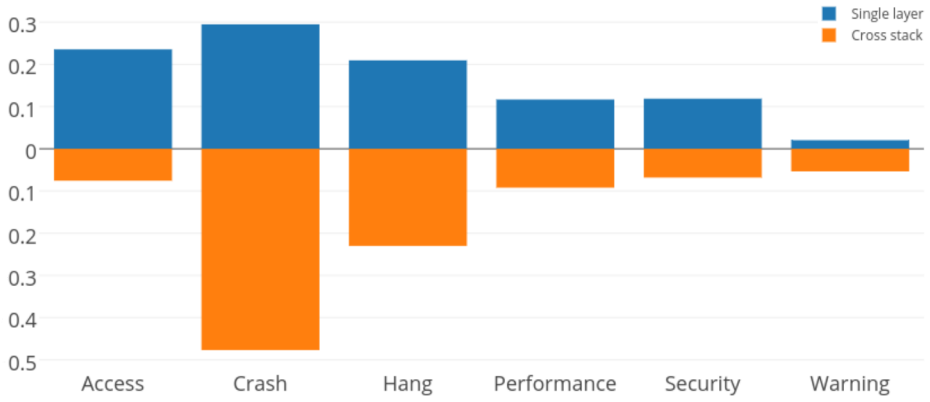


Figure 8.3 *Impact of single-layer errors vs. CsCEs.*

scripts or backups when operating the web application. For the same reasons as above, when restricting the analysis to production errors, the DevOps CsCEs grow to 44% compared to 26% for single-layer.

Conclusion: Cross-stack configuration errors have a severe impact compared to single layer errors, due to the high percentage of crashes they are responsible for, especially in the production environment. Due to this severe impact, sysadmins need automated support to debug and resolve CsCEs.

8.3.3 Effort to Solve Cross-stack Configuration Errors

In terms of effort to understand or fix configuration error, we did not find any statistically significant difference between single-layer and CsCEs for the number of comments on a question (Wilcoxon; p-value of 0.041), of proposed answers (p-value of 0.0274), of hours until a question took to be answered (p-value of 0.1297), of options discussed before finding the real culprit (median of 2, with a p-value of 0.3703), nor of comments on the provided answers (p-value of 0.430).

We hypothesize that the number of comments and answers are related to how well people expressed their problem and the forum members' experience. For example, we found that 25% of the questions was answered by the original poster, typically after a very long time. This could explain the difference in median time to answer a CsCE question of 2.33 hours for CsCEs compared to 1.7 hours for single-layer errors (mean of 485.1 and 253.9 hours, resp.).

Conclusion: While the literature reports that finding a single-layer misconfigured option is a hard and time-consuming task [95, 185, 215], we found that **CsCEs are at least as hard**

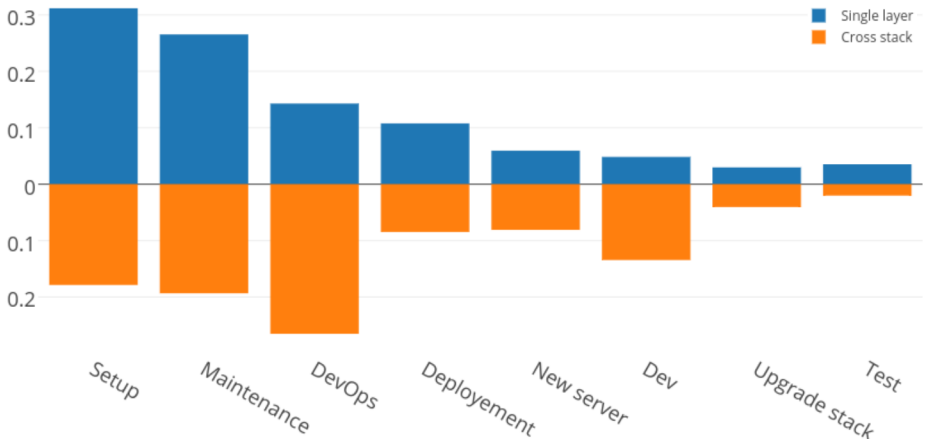


Figure 8.4 *When do single-layer and CsCEs occur?*

and time-consuming to resolve. This time could be reduced with automated support for resolving CsCEs.

8.3.4 Complexity of Cross-Stack Configuration Resolution

Whereas we found that **single-layer errors** have a relatively less severe impact than CsCEs, they seem to **require significantly more options** (p-value of $4.847e^{-05}$) **to be changed to fix them** than CsCEs (median of 2 vs. 1). Manual analysis showed that around 20 WP single-layer options always need to be changed together. Common cases are the options "WP_HOME" and "WP_SITEURL" [34] for WP, or options managing URL redirections and permissions for Apache (like "RewriteRule" and "RewriteCond" [5]).

While PHP interpreter errors were the most common (56%), 5% of the WP and DR CsCEs were caused by options assigned all the way inside the OS layer. We also found a non-negligible percentage of errors coming from the other layers, i.e., the web server (27%) and the database server (12%). The decreasing percentage from top to bottom layers is typical for a layered architecture, where layers mostly talk with their neighbours only. Furthermore, the 5% of CsCEs originating from the OS is a lower bound, as this study only considers CsCEs related to file system-related OS options, e.g. [17]). In future work, we plan to consider other OS options as well.

In 15% of the cases the user did not have access to one or more faulty configuration files, and hence had to override the misconfigured options by modifying them within the source code of the web application. Such overrides are risky, potentially causing additional conflicts. One example we found [35] showed a user modifying the "memory_limit"

option in the PHP-interpreter configuration file, but his modification did not work because the same option is overridden by the WP source code via the function "ini_set()".

Conclusion: CsCEs require to change options not only in the top layer of a stack, but also in deeper ones (all the way down to the OS), not all of which are open for change by the user.

8.4 Methodology for Identifying Cause of CsCE

This section presents our modular algorithm to recommend the configuration option responsible for a CsCE. We first discuss the slicing program analysis technique at the core of the algorithm, followed by important preliminary concepts.

8.4.1 Backward Slicing

Backward slicing is a program analysis technique used to find the statements that affect a given variable ("seed") used on a particular line of a program [198]. The line of code and seed together form the so-called "slicing criterion". Backward slicing is typically used to analyze, debug, and understand a program, since it reduces the scope of the program to only those statements impacting a targeted seed. Since its introduction by Weiser et al. [198], slicing techniques have seen applications in many domains [75]. Here, we use both static [198] and dynamic [37] slicing techniques.

To illustrate static slicing, let's consider the example code in subsection 8.5a and the resulting backward slice for the criterion (line 11, "higher") in subsection 8.5b. As shown, backward slicing starts from the targeted line, then goes backwards through the code to find all lines on which the seed variable "higher" is modified, and recursively repeats this for all variables and function calls whose value is used to calculate "higher". The resulting static slice is a compilable part of the original program that contains all statements required to calculate (i.e., that impact) the value of "higher" in line 11.

Dynamic slicing (subsection 8.5c) only analyzes the statements that were executed during a specific execution of the system, ignoring all other statements. For example, for an if-condition, it only analyzes the executed branch of an if-condition, ignoring the second branch. Dynamic slicing is better suited than static slicing to deal with reflective function calls, event handlers and dynamic file includes [84]. Furthermore, it reduces the size of the slice and hence the amount of source code to be analyzed, scaling better to larger systems. On the other hand, it requires invasive instrumentation to obtain the necessary dynamic data and concrete scenarios to run.

```
1  int HigherValue (int a , int b) {
2   String higher = "Result : ";
3   int result = 0;
4   if (a > b) {
5    result = a;
6    higher += "'a' is higher than 'b'";
7   } else {
8    result = b;
9    higher += "'b' is higher than 'a'";
10  }
11  write (higher);
12  return result;
13 }
```

Original code.

```
1  int HigherValue (int a , int b) {
2   String higher = "Result : ";
3
4   if (a > b) {
5
6    higher += "'a' is higher than 'b'";
7   } else {
8
9    higher += "'b' is higher than 'a'";
10  }
11  write (higher);
12
13 }
```

Static backward slice.

```
1  int HigherValue (int a , int b) {
2   String higher = "Result : ";
3
4   if (a > b) {
5
6    higher += "'a' is higher than 'b'";
7   }
8
9
10
11  write (higher);
12
13 }
```

Dynamic backward slice.

Figure 8.6 *Static vs. dynamic slicing for the criterion (line 11, "higher").*

8.4.2 Cross-stack Slice Dependency Graph

In subsection 8.2.1, we defined a stack as an acyclic graph in which edges represent dependencies between adjacent layers. In practice, such dependencies correspond to "physical links", i.e., some kind of mapping between resources used (e.g., function called, variables accessed or files read) in a layer and the definition of those resources (e.g., function definition, variable name or file name) in the layer below. Such physical links can be based on naming conventions, configuration files or could simply be hardcoded.

For example, in the LAMP stack, a Wordpress plugin would call functions in Wordpress using regular PHP function calls, while PHP primitives, global variables or standard library functions would be called from Wordpress using a PHP-to-C naming convention (e.g., function "is_uploaded_file" in Wordpress could map to "is_uploaded_file" in the PHP interpreter). MySQL could be called from the PHP interpreter via its official C API or via SQL queries.

In each layer, we can summarize the results of either dynamic or static slicing in the form of "slice dependency graphs". For each expression occurring in the slicing results, there is a

corresponding node in the layer's slice dependency graph that will have dependencies (edges) to the previous expression in the layer's slice. If the expression is preceded by if/else or switch/case conditions, it will depend on each condition, see the solid edges for the PHP layer in Figure 8.9.

Furthermore, as is typical for slicing techniques, if the expression is a function call, it will depend on all return statements of the called function, while the function definition will depend on all function calls to it. In case of access to a global variable, the node will depend on the last statement modifying that variable. Recursion typically is eliminated [75].

Finally, to integrate the slicing results across all layers of a given stack S, we introduce the notion of a "cross-stack slice dependency graph" $G =< N, E >$, where $< N, E >= \bigcup_{i,l} < N_i^l, E_i^l > \cup < \phi, E^{phys} >$, where $< N_i^l, E_i^l >$ is the slice dependency graph of a given component i of some layer l. A layer can have more than one slice dependency graph if it contains parts (e.g., components) that are independent from each other, for example different Wordpress plugins or MySQL stored procedures.

The key enabler for G is E^{phys}, which is the set of edges derived from the physical links that map an expression (node) in a graph of a given layer to an expression (node) in a graph of an adjacent, lower layer. E^{phys} maps a function definition in a layer to its calls in other layers, function calls to the return node(s) of the called function defined in another layer[1], and a variable access to its last modification in other layers.

In other words, using the physical links of the stack, a cross-stack slice dependency graph stitches together the individual slice dependency graphs of each layer into one giant graph, as illustrated in Figure 8.9.

8.4.3 CsCE Root Cause Recommendation

Our modular CsCE root cause recommendation algorithm ranks configuration options from most likely cause of a CsCE to least likely. Its main contribution is that it integrates existing static or dynamic slicing techniques applied on different layers or even individual components of a layer, instead of requiring a customized slicer per stack. Indeed, given the huge variety in programming languages and technologies in the layers of a stack, and the even larger flexibility in dependencies between them, a custom approach simply would not be feasible. Instead, our approach uses the existing slicing techniques in each layer to generate the slice dependency graphs, stitches them together in a cross-stack slice dependency graph, then traverses that graph to recommend options.

[1] In case of dynamic slicing, we know exactly which return node was used, but for static slicing we need to map to all possible return nodes.

```
1  culpritOptions: string[];
2  nodesProcessed: set;
3  errLine=findErrorMsgLine(errorMsg);
4  crossStackDepGraph=sliceAndCreateGraph(errLine);
5  errNode=graphNode(errLine,crossStackDepGraph);
6  for n: node in breadth-first traversal of crossStackDepGraph starting from errNode do
7  |  if n ∉ nodesProcessed then
8  |  |  nodesProcessed.add(n);
9  |  |  for o: configuration_option used by n do
10 |  |  |  culpritOptions.append(o);
11 |  |  end
12 end
```

Algorithm 1: *CsCE root cause recommendation algorithm.*

The main algorithm of our approach is presented in algorithm 1, taking as input an error message generated by a given software stack. First (line 3), it will try to find the line of code printing the error message using regular expressions. If this cannot be automated, or if no explicit error message is provided for a CsCE, almost always some symptom of the CsCE can be identified manually (for example an infinite loop, failing connection or a particular GUI element involved in the CsCE). In such cases, one could substitute *errLine* on line 3 by the manually identified line, which can then be used on line 4 to perform slicing on each layer, generate the layers' individual slice dependency graphs, then construct the unified cross-stack slice dependency graph.

The essence of the algorithm is a breadth-first traversal of the cross-stack slice dependency graph starting from the error message node *errNode* (line 6). Starting from *errNode*, we navigate the node's backward slice in breadth-first fashion along the node's edges in the dependency graph, and check each such dependent node for manipulation of a configuration option. If so, we add it to *culpritOptions*. To avoid visiting the same subgraph more than once, we use a cache to mark the visited nodes (line 7). Finally, when following an edge from a function definition to a call, the breadth-first iteration on line 6 of the algorithm will ignore the edge from the call to the function's return node. For example, after going from the *php_bar()* definition to the *php_bar()* call (white node), the algorithm will not return back to *return 0*.

We use breadth-first traversal for the graph navigation, since this will bring us first to the configuration options closest to the error message. Those are the options traditionally [46] considered to have the highest likelihood of being the cause of a configuration error. Options at the same distance from the error message node will be returned in a random order. Since

the unified cross-stack slice dependency graph spans across all layers, the algorithm traverses both lower and higher layers to find the cause of a CsCE.

Some optimizations are possible. For example, when performing the slicing and creation of slice dependency graphs on line 4, the slicing results of a given layer L_i could be used to limit the code that should be sliced in the next layer below L_{i+1}. Indeed, using a call graph, one could filter out the functions from L_{i+1} that could never be reached from L_i via the physical links between the layers. This works both for dynamic and static slicing. In Figure 8.9, the white nodes could be ignored by the algorithm, as they are not reachable from the definition of *php_bar()*, which was called from the top layer. A second optimization would be to stop the traversal on line 6 as soon as enough unique options have been appended to *culpritOptions*.

8.5 Empirical Evaluation

8.5.1 Setup of Empirical Evaluation

We evaluate algorithm 1 on 36 real CsCEs that we reproduced in a local LAMP environment to addresses the following research questions:

RQ1 How accurate is our approach?

RQ2 How fast is our approach?

Data Selection

Our evaluation considers a LAMP stack with the top three layers of Figure 8.1, i.e., the plugins layer, web app (Wordpress/Drupal) and the PHP-interpreter. The 36 evaluated CsCEs belong to three data sets (Table 8.4):

- **WP Set**: CsCEs occurring when using Wordpress that are due to a misconfigured option in the PHP interpreter.
- **DR Set**: The same, but when using Drupal.
- **Plugins Set**: CsCEs that occur during the use of a Wordpress plugin, and that are due to a misconfigured option in Wordpress or in the PHP interpreter.

To obtain the WP and DR Sets, we used three iterations:

Table 8.3 *The subject systems used in our evaluation.*

	Subject	Versions	#LOC	#options
1	WP	3	89,136	307
2	WP	4	131,044	316
3	DR	7.x-3.38	311,935	222
4	DR	7.5	112,961	153
5	DR	4.1.0	3,105	4
6	Woocommerce	2.4.8	57,725	132
7	Hyper Cache	3.2.3	1,098	32
8	UpdraftPlus	1.11.15	92,616	120
9	WP Super Cache	1.4.6	5,437	20
10	WP Photo Album+	6.3.10	43,587	739
11	NextGEN Gallery	1.6.1	8,599	73
12	Sitemap XML	1.5.0	202	9
13	PHP-interpreter	5.3.29	661,943	639
	#preprocessed LOC		5,057,274	

- First, during our qualitative analysis of section 8.3, we analyzed the 201 CsCEs that are due to a misconfigured option in the PHP-interpreter and occurred while using Wordpress or Drupal, in order to identify those that could be reproduced locally with the available configuration information. We also added two additional CsCEs encountered during our previous experiments [162]. We ended up with 92 configuration errors that, in theory, should be reproducible. This step was part of our qualitative study.

- The second iteration was the most tedious and time-consuming, as we tried to reproduce each of the 92 CsCEs on our local LAMP setup. It soon became clear that often crucial information required to reproduce an error was missing from a forum conversation. The major challenges of CsCEs, in particular the need to understand each layer's configuration options and their interactions, as well as missing version numbers of some of the layers, made this iteration quite painful and tedious. In the end, after substantial trial-and-error, we were able to reproduce 43 of the 92 configuration errors.

- In the third iteration, we filtered out CsCEs with similar symptoms and caused by the same configuration option, ending up with 29 distinct CsCEs.

To obtain the Plugins Set, we randomly selected errors from the official Wordpress forum and StackOverflow, using the configuration options of Wordpress and the PHP interpreter as keywords, then manually identifed whether the problem is related to a Wordpress plugin. By following steps 2 and 3 above, we obtained 7 new reproducible errors out of 23.

As shown in Table 8.3, our evaluation eventually considers 13 layer instances: WP (2 versions), DR (3 versions), 7 WP plugins, and the PHP interpreter (1 version). The number of configuration options in these instances ranges from 4 to 739, and each instance has a medium code size, ranging up to 661,943 SLOC [203]. When analyzing a full stack consisting of one WP/DR instance, one or more WP plugins and the PHP-Interpreter, the total number of options considered in the evaluation of a CsCE is the sum of options of all layers. The evaluated plugins are amongst the top 50 most popular WP plugins, with two of them amongst the top 5.

Implementation of Approach

As described in section 8.4, our approach combines existing static or dynamic slicing tools inside each layer. To deal with the complexities of the dynamic PHP language [84], we performed dynamic slicing on PHP-based layers (plugins and WP/DR layer), while we used a static slicing approach on the C-based PHP-interpreter. Our prototype implementation respectively uses our dynamic PHPSlicer [159] and static C BackSlicer [158] tools. Nguyen et al. [129]'s static slicer for PHP could be an alternative for PHPSlicer, yet it does not consider all dynamic PHP features (like dynamic includes, variable of variables, ...).

To build the database of physical links between layers, we manually analyzed the WP, DR and PHP interpreter source code and found two main kinds of physical links:

- Function call to a function implemented in the PHP interpreter. For example, the basic PHP function "move_uploaded_file" implemented in "ext/standard/basic_functions.c" of the PHP-interpreter can be called from any higher PHP layer by the same name "move_uploaded_file". The PHP interpreter offers more than 2,000 such functions to web apps [28].
- Superglobal variables, i.e., variables modified in the PHP interpreter that can be used from any web app. For example, the superglobal variable "$_REQUEST", when used in a web app, actually calls the function "php_auto_globals_create_request" in the file "main/php_variables.c". 9 such variables exist [30].

We implemented algorithm 1 in Java, exploiting the physical links above, and calling out to PHPSlicer and BackSlicer for the actual slicing. We optimized the execution time and accuracy of the static C slicing using the dynamic slicing results of the PHP-based layers, as explained in subsection 8.4.3. This improvement was essential to make the C slicing scale to the preprocessed version of the PHP-interpreter code base.

Evaluation of Performance

To evaluate the performance of our approach, we used two main metrics. For RQ1, we considered the rank of the reported misconfigured option in the output of algorithm 1. The lower this rank, the better, since this indicates that a user needs to try out less options before finding the root cause option.

For RQ2, we measure the total execution time of our algorithm. Again, the lower this metric, the better. We did not count the time to (1) instrument the web application for dynamic slicing purposes, and to (2) re-run the system to reproduce the error on the instrumented version, since (1) is done only once and (2) never surpassed 60 seconds. Table 8.4 summarizes our answers to the two research questions.

RQ1: How accurate is our approach?

Answer: Our approach has a high accuracy for ranking the misconfigured options, ranking 32 configuration errors with an average rank of 2.18 and a median of 1, with only 4 errors unable to be ranked.

From the 32 configuration errors for which we are able to find the cause, in 28 cases we report the culprit option with a good ranking (1st or 2nd suggestion), and in one case with an acceptable ranking (5th position), while in only three cases a low ranking of 10 or 12 was obtained. However, we think that even if that ranking is not ideal, it is still much better compared to manual debugging.

Based on the distance-based ranking criterion of our algorithm (algorithm 1), we are able to report the misconfigured option as the first suggestion even if the distance between the print statement generating the error message and an access to the culprit option is large. For example, for the 3rd and 24th CsCEs we are able to find the misconfigured option as the first suggestion, even though the distance between the C-slicing criterion and the access is a distance of 13 slicing graph edges apart.

The case with a ranking of 5 corresponds to the 10th CsCE, where the distance is 9 slicing graph edges. Such a ranking is still acceptable in practice, since manually finding these options would be hard due to the median distance and the complexity of the PHP-interpreter source code.

In three cases, we were only able to report the culprit option as the 10th and 12th suggestion, due to the large number of configuration options used in the sliced source code, and the high slicing graph distance of 30 and 32 edges. Considering additional strategies for traversing the

cross-stack slice dependency graph or ranking the *culpritOptions* (e.g., [192]) in algorithm 1 could further improve the results.

Deeper analysis of our results showed that the 32 successful CsCEs could be divided into three groups. The "Narrow" group corresponds to the narrow definition of a software stack (subsection 8.2.2), where the code line printing the error message and the line with the culprit option belonged to different layers of the stack. The "Broad 1" group contains examples of the first broader interpretation of CsCEs, where the WP/DR layer generates the CsCE symptom and reads the value of the offending PHP interpreter option (via the function "ini_get()"). Finally, the "Broad 2" group contains examples of the second broader interpretation, where symptoms, misconfigured option, and access to the option's value really happened within the same (PHP interpreter) layer, yet (due to the "error_reporting" option being "on") the PHP interpreter's error messages showed up on the user-visible Wordpress web page, not just in the execution logs, causing confusion.

In total, out of the 32 CsCEs, 8 belonged to the "Narrow" group, 14 to "Broad 1" and 11 to "Broad 2". Note that narrow errors are impossible to detect by existing single layer approaches (see Section 8.2.3). The broad categories may be found by existing approaches, but only for higher layers, in which most of the functions (slice dependency graphs) tend to be connected by a call graph. The deeper one goes, the more functions (slice dependency graphs) are disconnected, since lower layers are called by higher ones to provide a specific service, then return. For example, the two subgraphs within the MySQL layer in Figure 8.9 are not connected directly. However, they are connected via the PHP layer graph. Without this cross-layer context, it is impossible to navigate between functions and hence apply existing single-layer approaches. Of course, even if all functions in all layers would be connected to each other, it is still impossible to predict ahead of time whether the culprit option of a configuration error really belongs to the same layer as the error symptom. Hence, even for "Broad 2", our generic approach is the most pragmatic.

Finally, we also analyzed the four cases for which we were not able to rank the misconfigured option at all. The first reason our approach failed is when there was no concrete starting point for the slicing (14th, 35th, and 36th CsCEs). For the first two of these errors, the culprit configuration option disabled the execution of plugin PHP code altogether (option "short_open_tag" was set to "off"). For the 36th case, the user is redirected to the login page after trying to access the admin panel, without any error message. In general, unexecuted code or lack of symptom is a problem for all approaches.

The second reason, preventing us from ranking the 13th error case, is that the error message was shown in the browser by JavaScript code after an Ajax call. Since our work currently

does not consider JavaScript and its asynchronous calls, we plan to combine our approach with existing work [39, 129] that analyzes client source code (HTML, JS).

Note that, similar to related work on debugging configuration errors, we also assume that an error is already classified as being a configuration issue. Our technique can be complemented by the approach of Wen et al. [200] to first classify an error as configuration or non-configuration error.

RQ2: How fast is our approach?

Answer: The errors are reported within minutes, which makes our approach practically useful for users.

Dynamic slicing requires an execution trace, which can be generated by instrumenting an application and executing it. The instrumentation took 3.57 seconds in the best case, and 21.62 minutes in the worse case, with this time typically related to the size of the instrumented layer (in SLOC). Even when the instrumentation takes around 21 minutes, this needs to be done only once, after which the instrumented version of the system can be deployed and made available by customer service to all troubleshooting users, for example by manipulating the DNS server or load balancer [49].

To generate the execution trace from the instrumented version, a user needs to reproduce the error on the instrumented version of their web app. Although the instrumentation makes the web app slower, from our evaluation the error reproduction did not require more than 60 seconds on the evaluated errors.

After the error reproduction, one has to execute our algorithm, which takes between 35.77s and 553.18s (median of 230.10s) to perform the slicing and find the misconfigured option. Note that without the optimization that reduces the scope of slicing for lower layers based on the slicing results of higher layers, the C slicer sometimes would not finish, hence we considered this optimization crucial to make the approach compatible with static slicing of large layers. Hence, even if some steps can still be improved in future work, our approach is not only accurate but also fast enough for practical usage.

Since, to the best of our knowledge, we are the first to propose a generic approach to debug CsCEs, we compared our approach to a manual search in an online forum like StackOverflow, ServerFault or StackExchange, based on the qualitative study of Section 8.3. Without considering the time required to test a proposed answer and to discuss it via comments, one has to wait a median of 2.433 hours to get the correct answer for misconfigured PHP-interpreter options, assuming the question was answered and the accepted answer also applies to other

people. Although we do not have precise numbers on unresolved forum errors, we did find that 35% of PHP-interpreter CsCE threads are answered by the original poster. This suggests that (a) asking a question online does not guarantee an answer and that (b) in cases where the original poster had to find the answer on her own, she took the effort to follow up on her own question. Instead, our approach is more likely to propose a relevant answer, and does this in a median of only 0.06 hours (current prototype).

8.6 Threats to Validity

8.6.1 Qualitative Analysis

Despite the effort spent on our qualitative study design, gathering and clustering data from Q&A forums, we identified several threats to validity. First, extracting data from StackExchange and its sub forums is subject to construct validity, since we used the configuration options as keywords retrieve discussion threads. Relevant discussions not mentioning explicit option names may have been missed.

Moreover, the data extracted is subject to potential threats to reliability due to the gamification characteristics of StackExchange [191]. Q&A participants compete for reputation points and badges, which could encourage them to guess which configuration option might be the root cause of a CsCE, introducing bias into our metrics (such as the number of options or layers discussed). Fortunately, questions and answers are voted upon by the community, filtering out guesswork.

Furthermore, we manually analyzed the discussions' text to cluster threads in categories, which could introduce subjectivity in the analysis. To counter this, we used two raters and a multi-iteration approach for card sorting, and we also analyzed a large data set of 2,387 threads.

Finally, regarding threats to external validity, we only considered LAMP-related discussions, and we combined the single-layer and CsCE data of the five analyzed layers to deal with data imbalance. Given its popularity in the field, we believe LAMP to be highly representative. Furthermore, the large number of single-layer and CsCE data analyzed, as well as the variety of observations that we made, provide us confidence about our results. Studies on other stacks and other LAMP web apps should be performed in the future.

8.6.2 Empirical Evaluation

Regarding the threats to external validity of our evaluation, we only analyzed Wordpress (plugins), Drupal and the PHP interpreter, hence we cannot generalize the results to other stacks, nor to lower layers such as Apache and operating systems. However, our outcomes show promising results, encouraging us to evaluate the approach as well on other stacks such as MEAN.

Regarding threats to internal validity, the number of analyzed and evaluated configuration errors is high compared to related work (Table 8.5), with the number of analyzed configuration errors twice the number of Yin et al.'s study [215], and the number of real evaluated error 50% higher than Dong et al. [68]. In future work, we aim to evaluate even more real CsCEs, although reproducing such errors is time-consuming.

Another threat to internal validity could be the fact that we did not evaluate our approach in cases where more than one option was misconfigured. However, we think that in such cases, users can fix an initial option using our tool, then re-execute their scenario to find the second misconfigured option, and so on. For future work, we aim at exploring such cases.

Finally, we only focus on errors that cause a system to crash or write out an error message. We are not focusing on misconfiguration errors that have an impact on the system's performance only. However, in section 8.3, we found that the majority of configuration errors exhibit a Crash or a Hang. For future work, we plan to consider other kinds of errors.

8.7 Conclusion

This paper empirically studied the impact, effort and fix complexity of cross-stack configuration errors (CsCEs) for 1,082 online configuration errors, showing that CsCEs are common and have a severe impact, even in production. We then proposed the concept of cross-stack slice dependency graph and an accompanying modular algorithm to recommend the culprit option of a CsCE by integrating the results of existing slicing algorithms. Empirical evaluation on 36 real CsCEs in a LAMP stack showed that the approach provides a good ranking in a minimal amount of time, and could be integrated into the workflow of online stack hosting. Future work should evaluate our approach on other stacks and additional, deeper layers.

```
1  //Wordpress [PHP]
2  if (Option5) {
3    bar();
4  }
5
6  //MySQL [C]
7  bool mysql_foo() {
8    return option3;
9  }
10
11 bool mysql_bar() {
12   return option4;
13 }
```

```
1  //PHP interpreter [C]
2  int php_foo () {
3    if (option1 == 10
4        && ! mysql_foo()) {
5      php_bar ();
6    }
7  }
8
9  int php_bar () {
10   if (option2
11       && mysql_bar()) {
12     print (error);//SYMPTOM
13   }
14   return 0;
15 }
```

Figure 8.8 *Example of a 3-layer LAMP stack.*

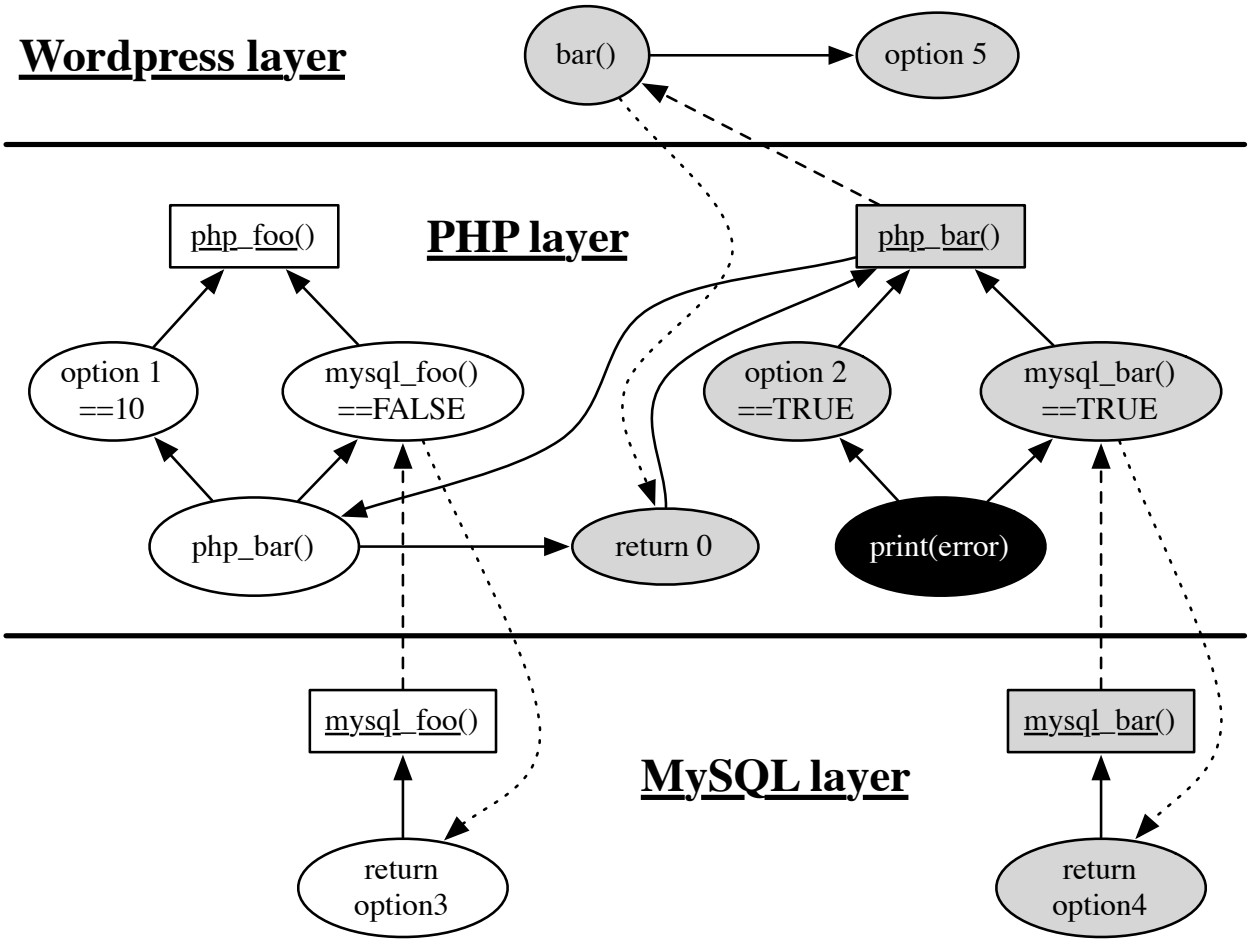


Figure 8.9 *Cross-stack slice dependency graph for Figure 8.8. Solid lines indicate slice dependencies, dashed lines physical links and dotted lines slice dependencies derived from the physical links. The black node is the start node, while the white nodes could be ignored for optimization.*

Table 8.4 *The evaluated CsCEs.*

Data		Error	System used	Misconfigured option	CsCE group	Rank	trace size	Time
WP Set	1	The maximum execution time allowed is exceeded	WP	max_execution_time (PHP)	Broad 2	1	1.1 MB	108.13s
	2	The folder's path used to save session data is incorrect	WP	session.save_path (PHP)	Broad 2	1	11.3 MB	553.18s
	3	The allowed memory size WP is requiring has been exceeded	WP	memory_limit (PHP)	Broad 2	1	3 KB	72.56s
	4	Unable to send mails	WP	sendmail_path (PHP)	Narrow	1	4.4MB	274.35s
	5	Web page inaccessible as its source code is outside the allowed path	WP	open_basedir (PHP)	Broad 2	1	-	57.79s
	6	upload_file_size has no effect on maximum file size to upload	WP	post_max_size (PHP)	Broad 1	2	9.3 MB	288.75s
	7	No additional database connections available to WP	WP	mysql.max_links (PHP)	Narrow	2	301 KB	54.46s
	8	File upload disabled	WP	file_uploads (PHP)	Narrow	2	8.2MB	58.98s
	9	Not able to upload a file	WP	max_file_uploads (PHP)	Narrow	2	9.9 MB	60.31s
	10	Incorrectly specified WP source code include path	WP	include_path (PHP)	Broad 2	5	811 bytes	67.54s
	11	The maximum number of form inputs a user can send is exceeded	WP	max_input_vars (PHP)	Broad 2	10	6.8 MB	369.91s
	12	Not able to upload a file	WP	post_max_size (PHP)	Narrow	10	6.5 MB	58.27s
	13	No results for Ajax query that exceeded the alotted time	WP	max_execution_time (PHP)	Narrow	-	7.3 MB	-
	14	Web app does not execute PHP code	WP	short_open_tag (PHP)	No symptom	-	-	-
Plugins Set	15	The plugin warns user from a lack of memory	Woocommerce	WP_MEMORY_LIMIT (WP)	Broad 1	1	24.7 MB	243.04s
	16	Not able to use caching features as it is disabled in WP	Hyper Cache	WP_CACHE (WP)	Broad 1	1	11.3 MB	234.64s
	17	Not able to use backup features in the plugin	UpdraftPlus	DISABLE_WP_CRON (WP)	Broad 1	1	14.9MB	238.42s
	18	Plugin disabled due to a WP option	WP Super Cache	permalink_structure (WP)	Broad 1	1	19.5MB	237.07s
	19	Not able to upload a file due to its large size	WP Photo Album+	upload_max_filesize (PHP)	Broad 1	1	13.1MB	236.01s
	20	Fail to upload a file due to a lack of memory	NextGEN Gallery	memory_limit (PHP)	Broad 1	1	12MB	235.73s
	21	Plugin reports error when compression is enabled in PHP-interpreter	WP Super Cache	zlib.output_compression (PHP)	Broad 1	1	11.7MB	235.24s
DR Set	22	DR reports that the option's value is very low	DR	max_execution_time (PHP)	Broad 1	1	87.9 MB	262.35s
	23	Idem to the last case	DR	memory_limit (PHP)	Broad 1	1	87.8 MB	256.65s
	24	DR crashes as the allowed memory limit is exceed	DR	memory_limit (PHP)	Broad 2	1	33 KB	73.90s
	25	Not able to load PHP extensions	DR	extension_dir (PHP)	Broad 2	1	43 KB	254.26s
	26	DR warns that option's value is incorrect	DR	magic_quotes_gpc (PHP)	Broad 1	1	228 KB	229.50s
	27	DR crashes due to a limit of execution time	DR	max_execution_time (PHP)	Broad 2	1	379 KB	78.10s
	28	Not able to upload a file as its size is not allowed	DR	upload_max_filesize (PHP)	Broad 1	1	6 MB	231.12s
	29	DR is reporting an error, while the value of that option is incorrect	DR	register_globals (PHP)	Broad 1	1	228 KB	230.34s
	30	The number of form inputs is limited	DR	max_input_vars (PHP)	Broad 2	1	13.5 MB	101.69s
	31	Restriction of files that can be used	DR	open_basedir (PHP)	Broad 2	1	-	35.77s
	32	Not able to enable a DR module due to a missed PHP extension	DR	extension (PHP)	Narrow	1	106.6 MB	63.90s
	33	upload_max_filesize has no effect on maximum file size to upload	DR	post_max_size (PHP)	Broad 1	2	1.3 MB	229.78s
	34	Not able to upload a file	DR	upload_tmp_dir (PHP)	Narrow	12	5.8 MB	56.44s
	35	DR shows its source code instead of executing it	DR	short_open_tag (PHP)	No symptom	-	-	-
	36	The user is redirected to the login page, without error message	DR	max_input_vars (PHP)	No symptom	-	4.7 MB	-

Table 8.5 *Comparison to evaluation in related work.*

Paper	#errors studied	#real errors evaluated	#random evaluated	cross-stack	# systems
Our work	1,082	36	0	yes	10
Zhang et al. [219]	394	8	0	no	6
Yin et al. [215]	546	0	0	no	5
Arshad et al. [43]	281	0	0	no	2
Dong et al. [68]	0	21	8	no	4
Attariyan et al. [46]	0	18	60	no	3
Zhang et al. [217]	0	14	0	no	5

CHAPTER 9 GENERAL DISCUSSION

We conducted in this thesis a set of empirical studies to better understand the process of configuration engineering as well as the challenges that are faced by practitioners. This allowed us to identify and evaluate a set of best practices to improve the quality of software configuration engineering. We discuss in this section our main findings and results of this thesis.

9.1 Software Configuration Challenges

9.1.1 Need for Approaches to Help Developers in Configuration Engineering Activities

While existing research literature points out that configuration errors are common, severe, and hard-to-debug problems, we took a step back to understand what can lead to such low quality, by reporting on problems that occur during the development cycle and that are faced by developers, which can have an impact on the software configuration engineering quality. For example, Xu et al. [212] found that users change only a few configuration options, which indicates that developers can simplify their software configuration by removing options that are not used by a software system's customers. However, as we found in our qualitative study, removing configuration options is challenging, because developers do not have a complete view about the impact of each configuration option in the source code, and hence removing one option can cause serious errors in the production.

In our qualitative study (Chapter 4), we identified 24 recommendations to help developers ensure a good configuration engineering quality, while many of these activities and recommendations are less covered by the research literature and requires further investigations.

9.1.2 Usage and Popularity of Configuration Frameworks

In the above study (Chapter 4), we used a qualitative approach to understand different challenges that practitioners face by interviewing developers and conducting a survey, and found that developers do not use existing configuration frameworks, and only 49% of surveyed developers use an existing configuration framework. Via a quantitative analysis of a large number of open source projects, Chapter 5 quantitatively confirmed this finding. We found that almost half of the projects that we analyzed does not use a configuration framework at all.

To better understand which frameworks developers tend to use, we conducted a quantitative analysis in which we studied the popularity of 11 different configuration frameworks, and found that developers tend to use simple and basic configuration frameworks, such as Java Properties and Preferences, although many more sophisticated and open source frameworks exist. That was also observed in our interviews (Chapter 4) and via our initial survey in Chapter 5.

9.2 Development of Best Practices to Improve the Quality of Software Configuration Engineering

In Chapter 4, one of the initial interviewed experts explained how his company reduces configuration problems by applying source code best practices to software configuration, while many other interviewed experts and surveyed participants who consider configuration as an external artifact do not apply such good practices on configuration options at all. This was also confirmed in our survey, in which we found that most surveyed participants just use tests as a quality assurance technique, 46% of participants do not use or respect a configuration naming convention, and 36% of them do not review configuration options at all.

Based on the principle of considering configuration as code, we derived a set of four principles inspired by our interviews that were prototyped in a framework called Config2Code. This study confirmed that using these principles, and considering configuration as code helps practitioners and developers improve their software configuration quality, with less effort (time required to achieve a typical configuration engineering task).

These four principles help users to add new options, refactor existing options by changing their names, their default values, and to remove options. These principles also help practitioners easily understand configuration options and find all their possible values.

However, the four principles did not help our 55 participants to debug configuration errors. We observed that our participants used a simple and basic approach to find which option is misconfigured. They just searched for options that might be related to the feature that is not correctly working, randomly changing their values, then test if the failure is fixed. Further analysis is required with different kinds of configuration errors (with and without an explicit symptom, performance, and security configuration errors) to observe how developers debug their configuration errors.

Furthermore, the Config2Code principles did not improve the reviewing of configuration patches. We think that this might be related to the patch that was reviewed by our user study participants, which just adds a new configuration option.

To the best of our knowledge, none of the existing configuration frameworks provides all four principles, although some of these principles are already implemented individually. For example, the configuration framework Constretto uses the mechanism of annotations, which enables configuration-as-code (our first principles), while it does not support the other 3 principles. Furthermore, no study empirically evaluated the principles, neither individually, nor together, on typical configuration engineering activities.

9.3 Cross-stack Configuration Errors

While Chapter 6 studies the potential of cross-stack configuration errors, Chapter 7 starts by studying the prevalence of such errors in real cases. We found in Chapter 6 that there is a significant potential of cross-stack configuration errors, as many configuration options are used not only by these options' layer but also by other layers. Chapter 7 confirmed the potential of cross-stack configuration errors, and how such errors are prevalent, severe, and hard-to-debug by analyzing real configuration errors in StackExchange fora.

We then proposed an approach (in chapter 7) to help debug cross-stack configuration errors. Our approach is modular and consists of composing different source code analysis techniques, each of which is applied to one layer. For example, one can apply a dynamic analysis technique on a PHP based web application and static source code analysis on the PHP-interpreter layer, and compose the results of both techniques via predefined physical links.

We conjecture that this approach can also be applied to cloud- and micro-services-based architectures, in which each web-service is consuming services of another web-service via a predefined protocol, such as SOAP or REST. One can apply a source code analysis technique to each web-service, then compose the results of the analysis of each web-service with the other ones.

We evaluated our approach by combining dynamic and static slicing techniques on the source code of the top 3 layers of the LAMP stack. While, in many cases, developers could not have access to the source code of one or multiple layers, we think that our approach can also accurately perform on a software system's byte-code, which we plan to evaluate in future work.

We were not able to find the culprit configuration for 4 cross-stack configuration errors, because we were not able to identify a starting point (an explicit error message) for these

options, as source code analysis techniques require a starting point. Using more advanced source code analysis techniques could resolve this limitation. For example, one can combine our approach with the approach of Lillack et al. [113, 114] to build configuration constraints for the whole LAMP stack, i.e., constraints between options that belong to different layers, then use the approach of Xiong et al. [210] to find which option does not respect these constraints and hence it is misconfigured. Such an approach would not require any source code starting point (explicit configuration failure symptom).

Our approach can also be combined with the approach of Attariyan et al. [44] to debug cross-stack performance configuration errors. According to such errors, one can have a performance degradation on a WordPress website due to an incorrect configuration in the web server or database, for example via the option *max_connections* that controls the number of allowed simultaneous connections.

CHAPTER 10 CONCLUSION

In this thesis, we conducted a set of studies to understand the process of engineering software configuration. We also studied and evaluated different strategies to improve software configuration engineering quality and address technical challenges that are faced by practitioners. Our research hypothesis stated:

> We hypothesize that (1) configuration engineering is constituted by a wide range of activities for which developers today face a variety of challenges. However, several of the technical challenges can be addressed by a careful selection of (2) source code best practices. For the specific activity of configuration debugging, (3) cross-stack configuration errors are more difficult to debug than single-layer configuration errors, however, they can be debugged effectively by a composition of source code analysis techniques.

Based on our results, we can confirm our research hypothesis. In particular, we found that practitioners face an important number of challenges at each level of the configuration engineering process and that a set of principles inspired from source code best practices can help address a set of important technical challenges related to the process activities. Then, in the context of the activity of configuration error debugging, we also found that cross-stack configuration errors are severe problems that require further investigation, and for which we proposed and evaluated an approach to debug cross-stack configuration errors.

10.1 Configuration Challenges and Recommendations

We studied software configuration from a practitioners' perspective and identified 9 activities that together form the process of configuration engineering, which starts from the creation of a new configuration option, how developers decide to create a new option, and which roles are involved in that creation. Another major configuration engineering activity is related to the maintenance of software configuration, which includes cleaning dead configuration options and changing an option's name or default value. An additional important activity is related to the access and uses of configuration options in the source code.

Related to these 9 activities, practitioners face 22 different software configuration challenges and problems. We found for example that practitioners do not plan the addition of new options, which are indeed added ad hoc and without any restrictions on who is responsible for adding new options. Since any developer can add new options, this can negatively impact the quality of software configuration engineering by having redundant and unclear configuration options. We also found that developers do not consider configuration options as regular source code, and hence do not apply best source code practices. An example of such ignored best practices is code review: many interviewed experts and surveyed participants ignore software configuration-related changes in their reviews, which obviously lets many problems slip through to the production environment.

Interviewed and surveyed participants provided 24 recommendations based on their own experience, with the goal of improving the quality of software configuration engineering addressing or minimizing the impact of the previously discussed 22 challenges. An example of these recommendations consists of delegating the creation and maintenance of software configuration to a few responsibles in an organization. This approach allowed the organization of one of the interviewed experts to reduce the complexity of software configuration in their code base, and minimize the number of redundant and unclear options. As for recommendations, we also discussed what should be included in the documentation of a software project, and how to organize options in the configuration file to make them easier to manipulate and understand.

Finally, we conducted a systematic literature review on run-time software configuration and found that most papers focus on debugging configuration errors and testing software configurations, which leave large interesting opportunities for future work. One important research direction that we identified is related to refactoring software configuration, which is considered risky, as developers generally do not recognize the impact of their software configuration options, and hence removing one option can have a negative impact that can propagate to production.

10.2 Usage of Configuration Frameworks

While it is highly recommended by practitioners to use configuration frameworks, we found in the previous study that developers do not adopt such frameworks. Therefore, we quantitatively studied the usage of these frameworks on Github open source projects and indeed confirmed our initial findings. We found that only 53% of analyzed projects use a configuration framework, where the most popular frameworks are the basic ones, i.e., Preferences and Java Properties, which are respectively used by 42% and 30% of the studied projects.

In addition to using basic frameworks, we found that 47.5% of projects using a configuration framework complement it with one to six additional configuration frameworks. This can have a negative impact on the maintenance of software configuration options.

Finally, we also analyzed the history of configuration framework usage in open source projects and analyzed the impact of these frameworks' features on the effort required by developers to maintain their software configuration. This can help practitioners choose a suitable framework for their own cases, by taking into consideration the most relevant factors.

10.3 Principles to Address Configuration Challenges

Developing good software configuration options does not only require testing different configurations of that software system but also preventing configuration problems from the development phases. The research literature mostly focuses on testing software configurations to avoid configuration errors, but many other aspects and practices can help improve software configuration engineering quality.

This is confirmed in our user study in Chapter 6, in which we proposed and evaluated the four principles discussed earlier and found that they are able to improve the correctness and time for 8 out of 11 typical configuration engineering tasks, including the creation and maintenance of software configuration options. We also found that these 4 principles do not have any negative impact on configuration engineering tasks.

10.4 Potential of Cross-stack Configuration Errors

While debugging configuration failures is an essential activity of the configuration engineering process, the research literature merely concentrates on single-layer and ignores cross-stack configuration errors. However, these errors might be prevalent and more difficult to fix. To confirm this end, we studied the potential prevalence of cross-stack configuration errors, by analyzing the top three layers of the LAMP stack. We found that WordPress plugins use 1.49% to 9.49% of WordPress database configuration options, while they are using 1.38% to 15.18% of all WordPress configurable constants, which are options stored in a configuration file.

Most surprisingly, we found that 78.88% and 85.16% of WordPress database and constant options are used by at least two different plugins. This indicates that almost all WordPress options are susceptible to introduce a cross-stack configuration error, as almost each WordPress option is used by at least two popular plugins, and also the modification of one option

could have an impact on multiple installed plugins at the same time. That can introduce hard-to-debug configuration errors, and increase the potential of cross-stack configuration errors.

10.5 Relevance and Debugging of Cross-stack Configuration Errors

While Chapter 7 confirmed an important potential of cross-stack configuration failures, Chapter 8 addresses their prevalence in real cases and compare their impacts to single-layer configuration errors. To this end, we studied 1,082 real configuration errors discussed in StackExchange fora. We found that cross-stack configuration errors have a severe impact compared to single layer configuration errors, as they mostly occur in production environments. We also found that cross-stack configuration errors require as much effort and time to be fixed as single layer configuration errors.

Then, we proposed a modular approach to debug such errors. This approach consists of using existing source code analysis techniques, such as dynamic or static slicing, in each layer, then to composes the results of each slicing analysis using physical links. These physical links can be built by relying on source code naming conventions. Our approach takes as input a symptom such as an error message and reports a set of options sorted from the most to the least likely to be misconfigured.

Finally, we evaluated our approach on 36 real configuration errors that occur in the top 3 layers of the LAMP stack. Our evaluation confirms that source code analysis techniques can help debug not only single layer configuration errors but also cross-stack configuration errors. We were indeed able to find the misconfigured option for 32 over 36 cases, in only a few minutes.

10.6 Future Work

Our research contributions open a wide range of opportunities for future work. The following sections discuss some of them.

10.6.1 Considering other Types of Software Configuration

Our research focuses on run-time configuration options, which are options whose value can be changed without recompiling or redeploying the software system. While there is an established volume of work on non-runtime configuration and variability, we believe that replicating our first study (Chapter 4) on this type of configuration can reveal interesting and

particular challenges and recommendations to that context. While our interviews (Chapter 4) cover a wide range of programming languages, we focused our survey only on Java runtime configuration options to avoid the bias of programming languages on our results. Thus, replicating our survey on other programming languages could reveal additional challenges and recommendations on software configuration engineering.

10.6.2 Resolving Additional Practical Configuration Challenges

As discussed in Chapter 4, our systematic literature review revealed that many research directions lack coverage by the literature. For example, we found that developers are not able to clean their software configuration options because they do not have a broad vision about option usage in the source code, and hence they are afraid of the side effects of configuration refactoring. We think that helping practitioners in this direction can reduce the complexity of software configuration options, as it will become easier for practitioners to find which options are not used anymore and hence need to be removed.

10.6.3 Evaluating other Best Practices on Configuration Quality

Chapter 6 proposes 4 principles that can fix many of the technical challenges identified in Chapter 4, such as helping practitioners remove dead options or preventing configuration errors by automatic validation of option values via checking of constraints. There is still a need to resolve organizational challenges, like the lack of communication between developers, for example. There is also a need to evaluate the impact of organizational best practices, such as pair-programming, on the quality of software configuration options.

10.6.4 Extending our Work on Debugging Cross-stack Configuration Errors

Our work on cross-stack configuration errors can be extended to different research directions.

Considering more Layers

We evaluated our modular approach only on the top three layers of the LAMP stack. We plan for our future work to extend our approach to consider other lower layers, such as the web server, database, and also the operating system layers.

Considering Other Stacks

We studied the LAMP stack for our evaluation of cross-stack configuration errors. In future work, we plan to extend and replicate our study on other types of stack architectures, such as the MEAN (MongoDB, Express.js, Angular.js, and Node.js) stack.

Evaluating our Approach on Cloud Applications

We believe that our modular approach not only applies to a stack architecture but can also help resolve configuration errors caused by multiple interacting web services. In this context, a user could get an error in her application, not due to a misconfiguration in that application, but due to a misconfigured option in a web service that it is using. Therefore, in future work, we plan to investigate such cloud application configuration errors.

Finding Correct Option Values

Our approach consists of debugging configuration errors and finding which options are misconfigured. However, one then needs to follow up using other strategies to find the correct values for these options. Xiong et al. [210] proposed an approach to find correct values for misconfigured options, however their approach relies on existing configuration constraints, which typically are not documented. One could recover such constraints using the approach of Lillack et al. [113, 114], which originally was proposed for single layer applications. Future work can rely on these approaches to build a constraint model across a stack of layers, and hence help find correct configuration values.

REFERENCES

[1] "Amazon configuration error," http://techgenix.com/aws-configuration-error-dow-jones/.

[2] "Amazon configuration error," https://www.esecurityplanet.com/network-security/configuration-error-exposes-almost-all-american-voters-personal-data.html.

[3] "Amazon configuration error," https://awsinsider.net/articles/2017/08/18/latest-amazon-s3-error.aspx.

[4] "Amazon misconfiguration," https://www.pcworld.com/article/226033/thanks_amazon_for_making_possible_much_of_the_internet.html.

[5] "Apache configuration error," http://serverfault.com/questions/557138/apache-dynamic-alias-based-on-sub-domain.

[6] "Apache web server." [Online]. Available: https://httpd.apache.org/

[7] "Apache zookeeper," http://zookeeper.apache.org.

[8] "Card sort of surveyed experts challenges," http://mcis.polymtl.ca/publications/2018/TSE/survey/ChallengesCardSort.html.

[9] "Card sort of surveyed experts recommendations," http://mcis.polymtl.ca/publications/2018//TSE/survey/BestPracticesCardSort.html.

[10] "Config2code exercise," https://www.dropbox.com/s/8nfpnzhny46t9gc/ex_config2code.pdf?dl=0.

[11] "Config2code exercise," https://www.dropbox.com/s/y88dneqqhsttkku/ex_preferences.pdf?dl=0.

[12] "Configuration error," https://dougseven.com/2014/04/17/ knightmare-a-devops-cautionary-tale/.

[13] "Configuration file," https://stackoverflow.com/questions/ 648246/at-what-point-does-a-config-file-become-a-programming-language.

[14] "Configuration problems in google," http://matt-welsh.blogspot.ca/2013/05/what-i-wish-systems-researchers-would.html.

[15] “Data:,” http://mcis.polymtl.ca/publications/2018//TSE/.

[16] “Drupal.” [Online]. Available: https://www.drupal.org/

[17] “Example of a system file permission error,” http://stackoverflow.com/questions/28843695/wp-cli-error-installing-plugins-themes-could-not-create-directory-permission.

[18] “Github api,” https://developer.github.com/v3/.

[19] “Interview card sort,” http://mcis.polymtl.ca/publications/2018//TSE/CardSort/.

[20] “J2ee.” [Online]. Available: http://docs.oracle.com/javaee/7/index.html

[21] “Jabref,” http://www.jabref.org/.

[22] “Lamp stack.” [Online]. Available: https://en.wikipedia.org/wiki/LAMP_(software_bundle)

[23] “Linux.” [Online]. Available: https://www.kernel.org/

[24] “Max input vars error,” http://stackoverflow.com/questions/21792891/wordpress-removes-my-menu-items-if-above-the-limit-of-90-menu-items.

[25] “Mean.” [Online]. Available: http://mean.io/#!/

[26] “mysql,” https://www.mysql.com/.

[27] “Original questions ideas of the interview,” http://mcis.polymtl.ca/publications/2018//TSE/Interviews/.

[28] “Php-interpreter.” [Online]. Available: http://php.net/

[29] “Study about surveys,” https://www.checkmarket.com/blog/survey-invitations-best-time-send/.

[30] “Superglobal variables.” [Online]. Available: http://php.net/manual/en/language.variables.superglobals.php

[31] “Survey answers,” http://mcis.polymtl.ca/publications/2018//TSE/survey/.

[32] “Survey questionnaire,” https://goo.gl/forms/cR7CkPgxcXxaddC12.

[33] “Wordpress.” [Online]. Available: https://wordpress.org/

[34] "Wp configuration error," http://stackoverflow.com/questions/17181148/wordpress-gets-404-not-found-error-when-entering-address.

[35] "Wp override problem," http://stackoverflow.com/questions/21680244/fatal-error-allowed-memory-size-of-268435456-bytes-exhausted-tried-to-allocate.

[36] E. K. Abbasi, M. Acher, P. Heymans, and A. Cleve, "Reverse engineering web configurators," in *Proceedings of the Conference on Software Maintenance, Reengineering, and Reverse Engineering*, ser. CSMR-WCRE'14, 2014, pp. 264–273.

[37] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *SIGPLAN Not.*, vol. 25, no. 6, pp. 246–256, 1990.

[38] B. S. Ahmed, T. S. Abdulsamad, and M. Y. Potrus, "Achievement of minimized combinatorial test suite for configuration-aware software functional testing using the cuckoo search algorithm," *Information and Software Technology*, vol. 66, pp. 13–29, 2015.

[39] S. Alimadadi, A. Mesbah, and K. Pattabiraman, "Understanding asynchronous interactions in full-stack javascript," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 1169–1180. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884864

[40] Ansible, "Core java preferences api," http://www.ansible.com/, Accessed March 08, 2018.

[41] "Apache commons configuration," https://commons.apache.org/proper/ commons-configuration/.

[42] S. Apel, D. Batory, C. Kästner, and G. Saake, "Feature-oriented software product lines: Concepts and implementation, berlin/heidelberg, 2013, 308 pages," ISBN 978-3-642-37520-0. URL http://www. springer. com/computer/swe/book/978-3-642-37520-0, Tech. Rep.

[43] F. A. Arshad, R. J. Krause, and S. Bagchi, "Characterizing configuration problems in java ee application servers: An empirical study with glassfish and jboss," in *Proceedings of the 24th International Symposium on Software Reliability Engineering*, 2013, pp. 198–207.

[44] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *Proceedings of the 10th Usenix Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 307–320.

[45] M. Attariyan and J. Flinn, "Using causality to diagnose configuration bugs," in *USENIX Annual Technical Conference*, Conference Proceedings, pp. 281–286.

[46] ——, "Automating configuration troubleshooting with dynamic information flow analysis," in *Proceedings of the 9th Usenix Symposium on Operating Systems Design and Implementation (OSDI)*, 2010, pp. 1–14.

[47] ——, "Automating configuration troubleshooting with confaid," *USENIX; login*, vol. 36, no. 1, pp. 27–36, 2011.

[48] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*, 1st ed. Addison-Wesley Professional, 2015.

[49] ——, *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[50] F. Behrang, M. B. Cohen, and A. Orso, "Users beware: Preference inconsistencies ahead," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 295–306.

[51] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki, "Variability modeling in the real: A perspective from the operating systems domain," in *Proceedings of the International Conference on Automated Software Engineering*, 2010, pp. 73–82.

[52] Q. Boucher, E. K. Abbasi, A. Hubaux, G. Perrouin, M. Acher, and P. Heymans, "Towards more reliable configurators: A re-engineering perspective," in *Proceedings of the 3rd International Workshop on Product Line Approaches in Software Engineering*, 2012, pp. 29–32.

[53] X. Bowen, D. Lo, X. Xin, A. Sureka, and L. Shanping, "Efspredictor: Predicting configuration bugs with ensemble feature selection," in *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pp. 206–213.

[54] R. P. Buse, C. Sadowski, and W. Weimer, "Benefits and barriers of user evaluation in software engineering research," in *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA'11, 2011, pp. 643–656.

[55] "Carbon," http://carbon.sourceforge.net/WhatIsCarbon.html.

[56] J. Cassoli, *Web Application with Spring Annotation-Driven Configuration: Rapidly develop lightweight Java web applications using Spring with annotations*, 1st ed. The address: CreateSpace Independent Publishing Platform, 10 2016.

[57] "Cfg4j," http://www.cfg4j.org/.

[58] H. Chen, G. Jiang, H. Zhang, and K. Yoshihira, "Boosting the performance of computing systems through adaptive configuration tuning," in *Proceedings of the ACM symposium on Applied Computing*, 2009, pp. 1045–1049.

[59] L. Christophe, R. Stevens, C. D. Roover, and W. D. Meuter, "Prevalence and maintenance of automated functional tests for web applications," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, Sept 2014, pp. 141–150.

[60] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 633–650, 2008.

[61] "Constretto," http://constretto.org/.

[62] B. Cornelissen, A. Zaidman, and A. Deursen, "A controlled experiment for program comprehension through trace visualization," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 341–355, 2011.

[63] B. K. Debnath, D. J. Lilja, and M. F. Mokbel, "Sard: A statistical approach for ranking database tuning parameters," in *procedings of the 24th International Conference on Data Engineering Workshop*, 2008, pp. 11–18.

[64] "Deltaspike," https://deltaspike.apache.org/.

[65] V. Denisov, "Overview of java application configuration frameworks," *International Journal of Open Information Technologies*, vol. 1, no. 6, pp. 5–9, 2013.

[66] Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus, "Managing web server performance with autotune agents," *IBM Systems Journal*, vol. 42, no. 1, pp. 136–149, 2003.

[67] Z. Dong, A. Andrzejak, D. Lo, and D. Costa, "Orplocator: Identifying read points of configuration options via static analysis," in *Proceedings of the 27th International Symposium on Software Reliability Engineering*, 2016, pp. 185–195.

[68] Z. Dong, A. Andrzejak, and K. Shao, "Practical and accurate pinpointing of configuration errors using static analysis," in *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, ser. ICSME'15, 2015, pp. 171–180.

[69] Z. Dong, M. Ghanavati, and A. Andrzejak, "Automated diagnosis of software misconfigurations based on static analysis," in *Proceedings of the International Symposium on Software Reliability Engineering Workshops*, 2013, pp. 162–168.

[70] S. Duan, V. Thummala, and S. Babu, "Tuning database configuration parameters with ituned," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 1246–1257, 2009.

[71] C. Elsner, D. Lohmann, and W. Schroder-Preikschat, "Fixing configuration inconsistencies across file type boundaries," in *37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011, August 30, 2011 - September 2, 2011*, ser. Proceedings - 37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011. IEEE Computer Society, Conference Proceedings, pp. 116–123. [Online]. Available: http://dx.doi.org/10.1109/SEAA.2011.26 http://ieeexplore.ieee.org/document/6068333/

[72] B. Eshete, A. Villafiorita, K. Weldemariam, and M. Zulkernine, "Confeagle: Automated analysis of configuration vulnerabilities in web applications," in *Proceedings of the 7th International Conference on Software Security and Reliability (SERE'13)*, 2013, pp. 188–197.

[73] L. Eshkevari, G. Antoniol, J. R. Cordy, and M. Di Penta, "Identifying and locating interference issues in php applications: The case of wordpress," in *Proc. of the 22nd ICPC*, 2014, pp. 157–167.

[74] S. Fouche, M. B. Cohen, and A. Porter, "Incremental covering array failure characterization in large configuration spaces," in *Proceedings of the 18th International Symposium on Software Testing and Analysis, ISSTA 2009*, pp. 177–187.

[75] K. Gallagher and D. Binkley, "Program slicing," in *Frontiers of Software Maintenance, 2008. FoSM 2008.*, Sept 2008, pp. 58–67.

[76] J. Gao, J. Guan, A. Ma, C. Tao, X. Bai, and D. C. Kung, "Testing configurable component-based software - configuration test modeling and complexity analysis," in *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering*, 2011, pp. 495–502.

[77] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Using feature locality: can we leverage history to avoid failures during reconfiguration?" in *Proceedings of the 8th workshop on Assurances for self-adaptive systems*, 2011, pp. 24–33.

[78] G. Gousios, "The ghtorrent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013, pp. 233–236.

[79] G. Gousios, V. Karakoidas, and D. Spinellis, "Tuning java's memory manager for high performance server applications," in *Proceedings of the 5th International System Administration and Network Engineering Conference SANE*, pp. 69–83.

[80] J. Gray, "Why do computers stop and what can be done about it?" in *Symposium on reliability in distributed software and database systems*. Los Angeles, CA, USA, Conference Proceedings, pp. 3–12.

[81] J. Guo, K. Czarnecki, S. Apely, N. Siegmundy, and A. Wasowski, "Variability-aware performance prediction: A statistical learning approach," in *Proceedings of the 28th International Conference on Automated Software Engineering*, pp. 301–311.

[82] S. Hamidi, P. Andritsos, and S. Liaskos, "Constructing adaptive configuration dialogs using crowd data," in *Proceedings of the 29th International Conference on Automated Software Engineering*, 2014, pp. 485–490.

[83] X. Han and T. Yu, "An empirical study on performance bugs for highly configurable software systems," in *Proceedings of the 10th International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM'16, 2016.

[84] M. Hills, P. Klint, and J. Vinju, "An empirical study of php feature usage: A static analysis perspective," in *Proc. of the 2013 Int'l Symposium on Software Testing and Analysis*, 2013, pp. 325–335.

[85] P. Huang, W. J. Bolosky, A. Singh, and Y. Zhou, "Confvalley: A systematic configuration validation framework for cloud services," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15, NY, USA, 2015, pp. 1–16.

[86] S. Huang, Y. Q. Lu, Y. Xiao, and W. Wang, "Mining application repository to recommend xml configuration snippets," in *Proceedings of the 34th International Conference on Software Engineering, ICSE'12*, pp. 1451–1452.

[87] A. Hubaux, Y. Xiong, and K. Czarnecki, "A user survey of configuration challenges in linux and ecos," in *Proc. of the 6th Int'l Workshop on Variability Modeling of Software-Intensive Systems*, 2012, pp. 149–155.

[88] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.

[89] D. Huning, C. Murphy, and G. Kaiser, "Confu: Configuration fuzzing testing framework for software vulnerability detection," *International Journal of Secure Software Engineering*, vol. 1, no. 3, pp. 41–55, 2010.

[90] "jconfig," https://sourceforge.net/projects/jconfig/.

[91] A. K. Jha, L. Sunghee, and L. Woo Jin, "Developer mistakes in writing android manifests: An empirical study of configuration errors," in *Procedings of the 14th International Conference on Mining Software Repositories (MSR'17)*, 2017, pp. 25–36.

[92] Y. Jiang and B. Adams, "Co-evolution of infrastructure and source code: An empirical study," in *Proceedings of the 12th Conference on Mining Software Repositories*, ser. MSR'15, 2015, pp. 45–55.

[93] Z. Jiaqi, L. Renganarayana, Z. Xiaolan, G. Niyu, V. Bala, X. Tianyin, and Z. Yuanyuan, "Encore: exploiting system environment and correlation information for misconfiguration detection," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 687–700, 2014.

[94] D. Jin, M. B. Cohen, X. Qu, and B. Robinson, "Preffinder: Getting the right preference in configurable software systems," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE 2014*, pp. 151–162.

[95] D. Jin, X. Qu, M. B. Cohen, and B. Robinson, "Configurations everywhere: Implications for testing and debugging in practice," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE'14, pp. 215–224.

[96] R. Johnson, "More details on today's outage," MathWorld–A Wolfram Web Resource, September 2010, https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919/.

[97] R. Kabacoff, *R in Action: Data Analysis and Graphics with R.* Greenwich, CT, USA: Manning Publications Co., 2015.

[98] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014, 2014, pp. 92–101. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597074

[99] A. Kapoor, "Web-to-host: Reducing total cost of ownership," Technical Report 200503, The Tolly Group, Tech. Rep., May 2000.

[100] S. Keele *et al.*, "Guidelines for performing systematic literature reviews in software engineering," in *Technical report, Ver. 2.3 EBSE Technical Report. EBSE*. sn, 2007.

[101] L. Keller, P. Upadhyaya, and G. Candea, "Conferr: a tool for assessing resilience to human configuration errors." IEEE, Conference Proceedings, pp. 157–66.

[102] A. Kenner, C. Kästner, S. Haase, and T. Leich, "Typechef: Toward type checking #ifdef variability in c," in *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development*, ser. FOSD '10. New York, NY, USA: ACM, 2010, pp. 25–32. [Online]. Available: http://doi.acm.org/10.1145/1868688.1868693

[103] M. Khan, Z. Huang, M. Li, G. A. Taylor, and M. Khan, "Optimizing hadoop parameter settings with gene expression programming guided pso," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 3, pp. 186–197, 2017.

[104] E. Kiciman and Y.-M. Wang, "Discovering correctness constraints for self-management of system configuration," in *Autonomic Computing, 2004. Proceedings. International Conference on*. IEEE, 2004, pp. 28–35.

[105] C. H. P. Kim, D. Marinov, S. K. D. Batory, S. Souto, P. Barros, and M. D'Amorim, "Splat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems," in *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13*, 2013, pp. 257–267.

[106] A. J. Ko, T. LaToza, and M. Burnett, "A practical guide to controlled experiments of software engineering tools with human participants," *Empirical Software Engineering*, vol. 20, no. 1, pp. 110–141, Feb 2015.

[107] P. S. Kochhar, D. Wijedasa, and D. Lo, "A large scale study of multiple programming languages and code quality," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 563–573.

[108] S. Krum, W. Van.Hevelingen, B. Kero, J. Turnbull, and J. McCune, *Pro Puppet*, 2nd ed. Apress; 2nd ed. edition, 07 2013.

[109] P. Lengauer and H. Mössenböck, "The taming of the shrew: increasing performance by automatic parameter tuning for java garbage collectors," in *Proceedings of the 5th international conference on Performance engineering*, pp. 111–122.

[110] W. Li, S. Li, X. Liao, X. Xu, S. Zhou, and Z. Jia, "Conftest: Generating comprehensive misconfiguration for system reaction ability evaluation," in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE'17, NY, USA, 2017, pp. 88–97.

[111] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1, May 2010, pp. 105–114.

[112] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, "Scalable analysis of variable software," in *Proc. of the 9th Joint Meeting on Foundations of Soft. Eng.*, ser. ESEC/FSE 2013, 2013, pp. 81–91.

[113] M. Lillack, C. Kästner, and E. Bodden, "Tracking load-time configuration options," *IEEE Transactions on Software Engineering*, pp. 1–1, 2017.

[114] M. Lillack, C. Kastner, and E. Bodden, "Tracking load-time configuration options," in *Proceedings of the 29th International Conference on Automated Software Engineering (ASE'14)*, 2014, pp. 445–456.

[115] D. Marijan, M. Liaaen, A. Gotlieb, S. Sen, and C. Ieva, "Titan: Test suite optimization for highly configurable software," in *Proceedings of the 10th International Conference on Software Testing, Verification and Validation*, 2017, pp. 524–531.

[116] S. McConnell, *Code Complete*, ser. DV-Professional. Microsoft Press, 2009. [Online]. Available: https://books.google.ca/books?id=3JfE7TGUwvgC

[117] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 643–654. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884793

[118] J. Meinicke, W. Chu-Pan, C. Kastner, T. Thum, and G. Saake, "On essential configuration complexity: Measuring interactions in highly-configurable systems," in *Proceedings of the 31st International Conference on Automated Software Engineering (ASE'16)*, 2016, pp. 483–494.

[119] X. Meng, P. S. Foong, S. Perrault, and S. Zhao, "5-step approach to designing controlled experiments," in *Proceedings of the International Working Conference on Advanced Visual Interfaces*, 2016, pp. 358–359.

[120] J. Mickens, M. Szummer, and D. Narayanan, "Snitch: Interactive decision trees for troubleshooting misconfigurations," in *In Proceedings of the 2007 Workshop on Tackling Computer Systems Problems with Machine Learning Techniques*, 2007.

[121] B. A. Mohammed Sayagh, Noureddine Kerzazi and F. Petrillo, "Software configuration engineering in practice interviews, survey, and systematic literature review," *Submitted to Transactions on Software Engineering (TSE)*, 2018.

[122] K. Morris, *Infrastructure as Code: Managing Servers in the Cloud*, 1st ed. The address: O'Reilly Media; 1 edition, 07 2016.

[123] Y. Murakami, E. Kagawa, and N. Funabiki, "Automatic generation of configuration manuals for open-source software," in *Proceedings of the 5th International Conference on Complex, Intelligent and Software Intensive Systems*, 2011, pp. 653–658.

[124] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Where do configuration constraints stem from? an extraction approach and an empirical study," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 820–841, Aug 2015.

[125] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: Static analyses and empirical results," in *Proc. of the 36th ICSE*, 2014, pp. 140–151.

[126] M. Nagappan, T. Zimmermann, and C. Bird, "Diversity in Software Engineering Research," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 466–476. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491415

[127] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Understanding and dealing with operator mistakes in internet services," in *OSDI'04*, Berkeley, CA, USA, 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251259

[128] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Exploring variability-aware execution for testing plugin-based web applications," in *Proc. of the 36th ICSE*, 2014, pp. 907–918.

[129] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Cross-language program slicing for dynamic web applications," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 369–380.

[130] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. Nguyen, "Dangling references in multi-configuration and dynamic php-based web applications," in *Proc. of the 28th Int'l Conf. ASE*, 2013, pp. 399–409.

[131] T. V. Nguyen, U. Koc, J. Cheng, J. S. Foster, and A. A. Porter, "Igen: Dynamic interaction inference for configurable software," in *Proceedings of the 24th International Symposium on Foundations of Software Engineering (FSE'16)*, 2016, pp. 655–665.

[132] A. J. Offutt and R. H. Untch, *Mutation 2000: Uniting the Orthogonal*. Boston, MA: Springer US, 2001, pp. 34–44.

[133] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, ser. USITS'03, 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251460.1251461

[134] T. Osogami and T. Itoko, "Finding probably better system configurations quickly," *Performance Evaluation Review*, vol. 34, no. 1, pp. 264–75.

[135] T. Osogami and S. Kato, "Optimizing system configurations quickly by guessing at the performance," in *Proceedings of the International Conference of Performance Evaluation Review*, 2007, pp. 145–156.

[136] H. Otsuka, Y. Watanabe, and Y. Matsumoto, "Learning from before and after recovery to detect latent misconfiguration," in *Proceedings of the 39th International Conference of Computer Software and Applications*, 2015, pp. 141–148.

[137] "Owner," http://owner.aeonbits.org/.

[138] S. K. Patel, V. Rathod, and J. B. Prajapati, "Performance analysis of content management systems-joomla, drupal and wordpress," *International Journal of Computer Applications*, vol. 21, no. 4, pp. 39–43, 2011.

[139] G. Perrouin, M. Acher, J. M. Davril, A. Legay, and P. Heymans, "A complexity tale: Web configurators," in *Proceedings of the 1st International Workshop on Variability and Complexity in Software Design*, 2016, pp. 28–31.

[140] "Play framework," https://www.playframework.com/.

[141] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Berlin Heidelberg, 2005. [Online]. Available: https://books.google.ca/books?id=lsX8_O_TRkEC

[142] X. Qu, "Configuration aware prioritization techniques in regression testing," in *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, 2009, pp. 375–378.

[143] X. Qu, M. Acharya, and B. Robinson, "Impact analysis of configuration changes for test case selection," in *Proceedings of the 22nd International Symposium on Software Reliability Engineering*, 2011, pp. 140–149.

[144] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: An empirical study of sampling and prioritization," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, 2008, pp. 75–85.

[145] R. Rabiser and D. Dhungana, "Integrated support for product configuration and requirements engineering in product derivation," ser. EUROMICRO'07, Piscataway, USA, 2007, pp. 193–200.

[146] A. Rabkin and R. Katz, "How hadoop clusters break," *Software, IEEE*, vol. 30, no. 4, pp. 88–94, July 2013.

[147] ——, "Precomputing possible configuration error diagnoses," in *Proceedings of the 26th International Conference on Automated Software Engineering*, 2011, pp. 193–202.

[148] ——, "Static extraction of program configuration options," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, 2011, pp. 131–140.

[149] M. Raghavachari, D. Reimer, and R. D. Johnson, "The deployer's problem: configuring application servers for performance and reliability," in *Proceedings of the 25th international conference on Software engineering*, 2003, pp. 484–489.

[150] M. T. Rahman, L.-P. Querel, P. C. Rigby, and B. Adams, "Feature toggles: A case study and survey," in *Proceedings of the 13th IEEE Working Conference on Mining Software Repositories (MSR)*, Austin, TX, May 2016, pp. 201–211.

[151] "Raihan's jconfig," https://github.com/prshreshtha/jconfig.

[152] V. Ramachandran, M. Gupta, M. Sethi, and S. R. Chowdhury, "Determining configuration parameter dependencies via analysis of configuration data from multi-tiered enterprise applications," in *Proceedings of the 6th International Conference on Autonomic Computing*, 2009, pp. 169–178.

[153] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter, "Using symbolic evaluation to understand behavior in configurable software systems," in *Proceedings of the 32nd International Conference on Software Engineering (ICSE'10)*, vol. 1, 2010, pp. 445–454.

[154] B. Robinson and L. White, "Testing of user-configurable software systems using firewalls," in *Proceedings of the 19th International Symposium on Software Reliability Engineering*, 2008, pp. 177–186.

[155] G. Rugg and P. McGeorge, "The sorting techniques: a tutorial paper on card sorts, picture sorts and item sorts," *Expert Systems*, vol. 22, no. 3, pp. 94–107, 2008.

[156] A. Sarkar, G. Jianmei, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-efficient sampling for performance prediction of configurable systems (t)," in *Proceedings of the 30th International Conference on Automated Software Engineering*, 2015, pp. 342–352.

[157] A. Sarma, G. Bortis, and A. van der Hoek, "Towards supporting awareness of indirect conflicts across software configuration management workspaces," in *Proceedings of the 22 International Conference on Automated Software Engineering*, ser. ASE'07, 2007, pp. 94–103.

[158] M. Sayagh and B. Adams, "Backslicer: A lightweight backward slicer," Ecole Polytechnique, Montreal, QC, Canada, Tech. Rep. 1, August 2016, http://mcis.polymtl.ca/~msayagh/TechnicalReports/MCIS-TR-2016-1.pdf.

[159] ——, "Phpslicer: Slicing dynamically typed programming languages - case study on php web apps," Ecole Polytechnique, Montreal, QC, Canada, Tech. Rep. 2, August 2016, http://mcis.polymtl.ca/~msayagh/TechnicalReports/MCIS-TR-2016-2.pdf.

[160] ——, "Data set of empirical evaluation," http://mcis.polymtl.ca/~msayagh/icse2017/DataSet/index.html.

[161] ——, "Videos," http://mcis.polymtl.ca/ msayagh/userStudy/Config2Code/.

[162] ——, "Multi-layer software configuration: Empirical study on wordpress," in *Proceedings of the 15th International Working Conference on Source Code Analysis and Manipulation (SCAM'15)*, 2015, pp. 31–40.

[163] M. Sayagh, Z. Dong, A. Andrzejak, and B. Adams, "Does the choice of configuration framework matter for developers? empirical study on 11 java configuration frameworks," in *Proceedings of the 17th International Working Conference on Source Code Analysis and Manipulation (SCAM'17)*, 2017, pp. 41–50.

[164] M. Sayagh, N. Kerzazi, and B. Adams, "On cross-stack configuration errors," in *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*, 2017, pp. 255–265.

[165] "SharedPreferences," https://developer.android.com/reference/android/ content/SharedPreferences.html.

[166] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR'16, 2016, pp. 189–200.

[167] H. Sheng, X. Yanghua, L. Yiqi, W. Wei, and W. Yu, "Xmlsnippet: A coding assistant for xml configuration snippet recommendation," in *Proceedings of the 36th Annual Computer Software and Applications Conference*, 2012, pp. 312–321.

[168] E. Shihab, Y. Kamei, B. Adams, and A. E. Hassan, "Is lines of code a good measure of effort in effort-aware models?" *Information and Software Technology*, vol. 55, no. 11, pp. 1981–1993, 2013.

[169] N. Siegmund, A. Grebhahn, S. Apel, and C. Kastner, "Performance-influence models for highly configurable systems," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*, 2015, pp. 284–294.

[170] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, 2012, pp. 167–177.

[171] R. Singh, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Optimizing the performance-related configurations of object-relational mapping frameworks using a multi-objective genetic algorithm," in *Proceedings of the 7th International Conference on Performance Engineering*, 2016, pp. 309–320.

[172] I. Sommerville, *Software Engineering*, 9th ed. USA: Addison-Wesley Publishing Company, 2010.

[173] C. Song, A. Porter, and J. S. Foster, "itree: Efficiently discovering high-coverage configurations using interaction trees," in *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, 2012, pp. 903–913.

[174] ——, "Itree: Efficiently discovering high-coverage configurations using interaction trees," *IEEE Transactions on Software Engineering*, vol. 40, no. 3, pp. 251–265, 2014.

[175] S. Souto and M. d'Amorim, "Time-space efficient regression testing for configurable systems," 2017.

[176] S. Souto, M. D'Amorim, and R. Gheyi, "Balancing soundness and efficiency for practical testing of configurable systems," in *Proceedings of the 39th International Conference on Software Engineering, ICSE'17, 2017*, 2017, pp. 632–642.

[177] "Spring framework," https://projects.spring.io/spring-framework/.

[178] Y.-Y. Su, M. Attariyan, and J. Flinn, "Autobash: improving configuration management with operating system causality analysis," in *ACM SIGOPS Operating Systems Review*, vol. 41. ACM, 2007, pp. 237–250.

[179] L. Sun, G. Huang, Y. Sun, H. Song, and H. Mei, "An approach for generation of j2ee access control configurations from requirements specification," in *Proceedings of the 8th International Conference on Quality Software*, ser. - International Conference on Quality Software. Inst. of Elec. and Elec. Eng. Computer Society, pp. 87–96.

[180] "Developer's survey via google forms," https://goo.gl/kXCQ7Q.

[181] J. Swanson, M. B. Cohen, M. B. Dwyer, B. J. Garvin, and J. Firestone, "Beyond the rainbow: Self-adaptive failure avoidance in configurable systems," in *Proceedings of the 22nd International Symposium on the Foundations of Software Engineering (FSE'14)*, 2014, pp. 377–388.

[182] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, "Feature consistency in compile-time-configurable system software: Facing the linux 10,000 feature problem," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 47–60. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966451

[183] R. Thonangi, V. Thummala, and S. Babu, "Finding good configurations in high-dimensional spaces: Doing more with less," in *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems*, 2008, pp. 1–10.

[184] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Comput. Surv.*, vol. 47, no. 1, pp. 6:1–6:45, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2580950

[185] X. Tianyin and Z. Yuanyuan, "Systems approaches to tackling configuration errors: a survey," *ACM Computing Surveys*, vol. 47, no. 4, pp. 1–41, 2015.

[186] J. Toman and D. Grossman, "Staccato: A bug finder for dynamic configuration updates," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 56. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[187] A. Tresch, "Java configuration," http://javaeeconfig.blogspot.de/2014/08/overview-of-existing-configuration.html.

[188] "Typesafe," https://github.com/typesafehub/config.

[189] S. Urli, M. Blay-Fornarino, and P. Collet, "Handling complex configurations in software product lines: A tooled approach," in *Proceedings of the 18th International Software Product Line*, 2014, pp. 112–121.

[190] F. van der Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer Berlin Heidelberg, 2007. [Online]. Available: https://books.google.ca/books?id=PC4LyoSNNAkC

[191] B. Vasilescu, A. Serebrenik, P. Devanbu, and V. Filkov, "How social Q&A sites are changing knowledge sharing in open source software communities," in *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, ser. CSCW '14, 2014, pp. 342–354.

[192] B. Wang, L. Passos, Y. Xiong, K. Czarnecki, H. Zhao, and W. Zhang, "Smartfixer: Fixing software configurations based on dynamic priorities," in *Proceedings of the 17th International Software Product Line Conference*. ACM, Conference Proceedings, pp. 82–90.

[193] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic misconfiguration troubleshooting with peerpressure," in *OSDI*, vol. 4, 2004, pp. 245–257.

[194] R. Wang, X. Wang, K. Zhang, and Z. Li, "Towards automatic reverse engineering of software security configurations," in *Proceedings of the 15th Conference on Computer and Communications Security*, 2008, pp. 245–255.

[195] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang, "Strider: A black-box, state-based approach to change and configuration management and support," *Science of Computer Programming*, vol. 53, no. 2 SPEC. ISS., pp. 143–164, 2004.

[196] C. Wei, W. Heng, W. Jun, Z. Hua, and H. Tao, "Determine configuration entry correlations for web application systems," in *Proceedings of the 40th Annual Computer Software and Applications Conference*, 2016, pp. 42–52.

[197] C. Wei, Q. Xiaoqiang, W. Jun, Z. Hua, and H. Xiang, "Detecting inter-component configuration errors in proactive: a relation-aware method," in *Proceedings of the 14th International Conference on Quality Software*, 2014, pp. 184–199.

[198] M. Weiser, "Program slicing," in *Proc. of the 5th ICSE*, 1981, pp. 439–449.

[199] M. Welsh, "What i wish systems researchers would work on," May 2013, http://matt-welsh.blogspot.ca/2013/05/what-i-wish-systems-researchers-would.html.

[200] W. Wen, T. Yu, and J. H. Hayes, "Colua: Automatically predicting configuration bug reports and extracting configuration options," in *Proceedings of the 27th International Symposium on Software Reliability Engineering*, 2016, pp. 150–161.

[201] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: a controlled experiment," in *2011 33rd International Conference on Software Engineering (ICSE)*, May 2011, pp. 551–560.

[202] ——, "Software systems as cities: A controlled experiment," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, New York, NY, USA, 2011, pp. 551–560.

[203] D. A. Wheeler, "Sloc count user's guide," 2004.

[204] A. Whitaker, R. S. Cox, and S. D. Gribble, "Configuration debugging as search: finding the needle in the haystack," in *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, 2004, pp. 77–90.

[205] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[206] H. Wu, L. Shi, C. Chen, Q. Wang, and B. W. Boehm, "Maintenance effort estimation for open source software: A systematic literature review," in *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, 2016, pp. 32–43. [Online]. Available: https://doi.org/10.1109/ICSME.2016.87

[207] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang, "A smart hill-climbing algorithm for application server configuration," in *Proceedings of the Thirteenth International World Wide Web Conference*, 2004, pp. 287–296.

[208] X. Xia, D. Lo, W. Qiu, X. Wang, and B. Zhou, "Automated configuration bug report prediction using text mining," in *Proceedings of the 38th Annual IEEE Computer Software and Applications Conference*, 2014, pp. 107–116.

[209] Q. Xiao, M. Acharya, and B. Robinson, "Configuration selection using code change impact analysis for regression testing," in *Proceedings of the International Conference on Software Maintenance*, 2012, pp. 129–138.

[210] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, "Generating range fixes for software configuration," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 58–68.

[211] Y. Xiong, H. Zhang, A. Hubaux, S. She, J. Wang, and K. Czarnecki, "Range fixes: Interactive error resolution for software configuration," *IEEE Transactions on Software Engineering*, vol. 41, no. 6, pp. 603–619, 2015.

[212] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs! : Understanding and dealing with over-designed configuration in system software," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering(ESEC/FSE'15)*, 2015, pp. 307–319.

[213] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proceedings of the 24th Symposium on Operating Systems Principles*, 2013, pp. 244–259.

[214] X. Xu, S. Li, Y. Guo, W. Dong, W. Li, and X. Liao, "Automatic type inference for proactive misconfiguration prevention," in *Proceedings of the 29th International Conference on Software Engineering and Knowledge Engineering*, 2017, pp. 295–300.

[215] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011, pp. 159–172.

[216] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar, "Context-based online configuration-error detection," in *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*. USENIX Association, 2011, pp. 28–28.

[217] S. Zhang, "Confdiagnoser: An automated configuration error diagnosis tool for java software," in *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, 2013, pp. 1438–1440.

[218] S. Zhang and M. D. Ernst, "Automated diagnosis of software configuration errors," in *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, 2013, pp. 312–321.

[219] ——, "Which configuration option should i change?" in *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, 2014, pp. 152–163.

[220] ——, "Which Configuration Option Should I Change?" in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 152–163. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568251

[221] ——, "Proactive detection of inadequate diagnostic messages for software configuration errors," in *Proceedings of the 24th International Symposium on Software Testing and Analysis*, 2015, pp. 12–23.

[222] Y. Zhang, J. M. Guo, E. Blais, and K. Czarnecki, *Performance Prediction of Configurable Software Systems by Fourier Learning*, 2015, pp. 365–373.

[223] W. Zheng, R. Bianchini, and T. D. Nguyen, "Massconf: automatic configuration tuning by leveraging user community information," in *Proceedings of the International Conference of Software Engineering Notes*, 2011, pp. 283–288.

[224] S. Zhou, X. Liu, S. Li, W. Dong, X. Liao, and Y. Xiong, "Confmapper: Automated variable finding for configuration items in source code," in *Proceedings of the 2nd International Conference on Software Quality, Reliability and Security-Companion*, 2016, pp. 228–235.

SURVEY QUESTIONNAIRE

Understanding Application Configuration

Application configuration allows to customize, tweak or select - without recompilation ! - the features and general behaviour (e.g., performance and security) of an application and the libraries it uses. This covers configuration of *application-specific* options as well as options of the *3rd party libraries/frameworks* used by this application (e.g., Log4J or JDBC), but *excludes* configuration of an application's infrastructure (e.g., Docker, Chef or Puppet).

Context: This survey is within the context of a PhD project at Polytechnic Montreal - Canada. Its goal is to understand how do you deal, create, and maintain configuration options in your application, and what are the best and bad practices that exist for the development of configuration options.

Note: All responses will be treated anonymously and confidentially, and will be aggregated to avoid singling out individuals.

Any comments or questions can be sent to: mohammed.sayagh@polymtl.ca
For more information about my work: mcis.polymtl.ca/~msayagh/

*Required

Demographical Questions

1. **How many years of software engineering experience do you have?** *

2. **What is your current role ?** *
   Mark only one oval.
   - ◯ Architect
   - ◯ Developer
   - ◯ Manager
   - ◯ Tester
   - ◯ Sysadmin
   - ◯ Other: _____

Your Experiences with Configuration Options

3. **On average, how many configuration options does your typical application have (application-specific) or use (3rd party library)?** *
Mark only one oval.

- ◯ No options
- ◯ <10
- ◯ 10<#<100
- ◯ 100<#<1,000
- ◯ 1,000<#<10,000

4. **Do you have personal experience with creating new configuration options?** *
Mark only one oval.

- ◯ Yes
- ◯ No

The Creation of Configuration Options

Configuration Mechanisms

5. **Where do you store your configuration options?** *
Tick all that apply.

- ☐ .properties file
- ☐ .xml file
- ☐ .json file
- ☐ .ini files
- ☐ In a table within the database used for the application
- ☐ In a table within a database dedicated to configuration
- ☐ Environment variables (e.g., Bash)
- ☐ LDAP or another directory service (e.g., Zookeeper)
- ☐ Application main arguments
- ☐ Other: _____

Configuration Options in the Source Code

6. **How do you access this configuration storage in the code?** *
   Tick all that apply.

   - [ ] Via an IO library to read the configuration storage.
   - [ ] Via a 3rd party framework dedicated to configuration
   - [ ] Via dedicated classes/functions that we developed for configuration access
   - [ ] Other: _____

7. **How are the values of application-specific configuration options available in the code?** *
   Tick all that apply.

   - [ ] As attributes/variables with the same name of its associated configuration option
   - [ ] From tables and/or maps containing configuration options and values
   - [ ] Via method/function accessors (for example getMyOption)
   - [ ] There is no clear mapping between options and code
   - [ ] Other: _____

8. **From where do you read your configuration options?** *
   Mark only one oval.

   - ◯ We read configuration options from a specific class/module
   - ◯ We read configuration options in different places in the code
   - ◯ Other: _____

Creation of new Configuration Options

9. **What is the most frequent profile of people adding new configuration options?** *
   Mark only one oval.

   - ◯ Certain expert developers
   - ◯ Any developer
   - ◯ Only architects
   - ◯ Architects and developers
   - ◯ Other: _____

10. **Why?** *

_____

_____

_____

_____

_____

11. **Is the creation of configuration options planned beforehand?** *
Tick all that apply.

- [ ] Yes, they are defined in the client specifications
- [ ] Yes, they are defined in the architecture specifications
- [ ] No, they are juste extracted from hardcoded constants
- [ ] No planning, options are added ad hoc
- [ ] Other: _____

12. **How do you decide configuration default values?** *
Tick all that apply.

- [ ] Client requirements.
- [ ] Architectural specifications.
- [ ] Discussion between developers and architects.
- [ ] Making tests until finding the optimal value for a parameter.
- [ ] Ad hoc choices. It is up to the developer to decide.
- [ ] Other: _____

13. **How do you communicate newly created options and their possible values to the team ?** *
Tick all that apply.

- [ ] Orally
- [ ] Chat support (like Slack or Hangout)
- [ ] By mail
- [ ] Via Wiki or a web platform
- [ ] Textual documentation
- [ ] Pull Request
- [ ] Other: _____

14. **Do you use any naming conventions to ensure a coherence between configuration option names?** *
Mark only one oval.

- Yes. We do have a naming convention, and all the developers respect it.
- Yes. We do have a naming convention, but not all the developers respect it.
- No. We do not have any configuration options naming convention.

Comprehension, Maintenance, and Debugging Configuration Errors

Debugging Configuration Errors

The following questions aim at understanding how you debug configuration errors, and what makes such debugging hard/easy. They are focusing on errors for both configuration types of this survey, i.e., application-specific and 3rd party configuration errors.

15. **How often do you face configuration errors?** *
Mark only one oval.

- Every day
- Often
- Sometimes
- Never

16. **How difficult is it to debug an error which is due to a misconfigured option?** *
Mark only one oval.

	1	2	3	4	5	
Very easy (One could identify the misconfigured option immediately)						Very hard (It could take the whole day to identify the misconfigured option)

17. **What artefacts do you use to fix a configuration error?** *
Tick all that apply.

- [ ] Log file
- [ ] Error message
- [ ] Documentation
- [ ] Debug the code
- [ ] Google the error
- [ ] Stack Trace
- [ ] Automatic tool to find misconfigured options
- [ ] Other: _____

18. **Where do you document configuration-related errors and their solutions?** *
Tick all that apply.

- [ ] In the commit message
- [ ] In a Wiki
- [ ] In a Bug report (like Bugzilla)
- [ ] In text document
- [ ] In the configuration file
- [ ] Nowhere
- [ ] Other: _____

Comprehension of Configuration Options

19. **Are you always aware of the impact of changing one of the application-specific configuration options?** *
Mark only one oval.

- ◯ Yes, I know the impact of all the configuration options.
- ◯ Yes, I know the impact of the majority of configuration options.
- ◯ Yes, I know the impact of few configuration options.
- ◯ Yes, I know the impact of only configuration used in the code I worked on.
- ◯ No, I'm not aware of the impact of changing options.

20. **Which artefacts do you use to understand application-specific configuration parameters?** *
Tick all that apply.

- [ ] Documentation.
- [ ] Ask a colleague
- [ ] Configuration option comments.
- [ ] Configuration name is clear enough to be understood.
- [ ] I understand configuration options from the source code
- [ ] I Google configuration names.
- [ ] Other: _____

Maintenance of Configuration Options

21. **Do the names or the default values of your application-specific configuration options change across time?** *
Mark only one oval.

- ◯ Yes. They change frequently
- ◯ Yes, they change but not frequently
- ◯ No, they are almost stable
- ◯ No, they never change. Once, they are created, they do not change anymore.

22. **What do you do with dead configuration options that are no longer useful?** *
Mark only one oval.

- ◯ We do not remove them, we keep them in the configuration files.
- ◯ Remove them from the configuration files, and store them in a databse.
- ◯ Remove them from the configuration files, and do not store them anywhere.
- ◯ Just Comment them in the configuration file.
- ◯ Other: _____

Quality Assurance of Configuration Options

Documentation of Options

23. **How would you rate the quality of the documentation (text documents or online documentation) of application-specific configuration options in the systems you participated in their developments ?** *
Mark only one oval.

	1	2	3	4	5	
Doesn't exist	◯	◯	◯	◯	◯	Excellent

24. **How would you rate the quality of the comments of configuration options (used within configuration files to explain the meaning and possible values of options)?** *
Mark only one oval.

	1	2	3	4	5	
Doesn't exist	◯	◯	◯	◯	◯	Excellent

25. **Please explain the factors that lead to that quality (your previous two answers) of comments and documentation** *

Code Review

26. **Do you consider configuration options during code review?** *
Mark only one oval.

- ◯ Yes, but we review only functional configuration options
- ◯ Yes, but we review only technical configuration options.
- ◯ Yes, we review both functional and technical configuration options.
- ◯ We do not review configuration options at all. We only review code.
- ◯ We do not review configuration options nor source code.

Approaches to Ensure the Quality of a System Configuration

27. **Which techniques do you use to assure the quality of configuration options?** *
Tick all that apply.

- [ ] We have tools to check correctness of the values
- [ ] We have a static analysis tool inspired by checkstyle to enforce configuration conventions.
- [ ] Default values are backed-up in a database to easily roll back.
- [ ] Security implications of configuration options are tested
- [ ] Test different configuration values
- [ ] Using containers (like Docker) to limit the impact of configuration options.
- [ ] Other: _____

28. **What could reduce the most the complexity of the configuration of a software project?** *
Tick all that apply.

- [ ] Reduce the number of configuration files.
- [ ] Allow only few people to modify configuration files.
- [ ] Allow only experienced people to modify configuration files.
- [ ] Reduce the configuration option dependencies
- [ ] Comment and document configuration options.
- [ ] Reduce the number of configuration options.
- [ ] Planning ahead the creation of options.
- [ ] Make every option responsibility of one specific person.
- [ ] Use a separate software to manage configuration options.
- [ ] Other: _____

Open and General Questions about Configuration

29. **Could you describe some important problems related to configuration (in terms of creation/management/maintenance or any other theme) that you faced during your current and/or previous experiences and how did you fix them ?** *

_____

_____

_____

_____

_____

30. **What could you suggest as 3 good practices and 3 bad practices to a new development team in order to have a good configuration quality and to avoid configuration problems?** *